MIT/LCS/TR-269

The Complexity of Concurrency Control for Distributed Databases

Paris C. Kanellakis

THE COMPLEXITY OF CONCURRENCY CONTROL FOR DISTRIBUTED DATABASES

by

Paris C. Kanellakis

Diploma, National Technical University of Athens
(1976)
S.M., Massachusetts Institute of Technology
(1978)

Submitted in Partial Fulfillment
of the Requirements for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology
September 1981

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science, September 1, 1981

Certified by ______________________________
Christos H. Papadimitriou
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

The Complexity of Concurrency Control for Distributed Databases

by

Paris C. Kanellakis

Submitted to the Department of Electrical Engineering and Computer Science
on September 1, 1981 in partial fulfillment of the requirements for
the Degree of Doctor of Philosophy

Abstract

This study is an analysis of the distributed version of database concurrency control. It provides concrete mathematical evidence that the distributed problem is an inherently more complex task than the centralized one.

The notions of transaction, concurrency, history, serializability, scheduler, etc., for centralized databases are now well-understood both from a theoretical and a practical point of view. A formal model for the case of distributed databases is presented. The transactions are partially ordered sets of actions, as opposed to the totally ordered straight-line programs of the centralized case. The scheduler is also a distributed program. Three notions of performance for a scheduler are studied and interrelated: (i) parallelism, (ii) the computational complexity of the decision problems that it has to solve, (iii) the cost of communication between the various parts of the scheduler. In fact the number of messages necessary and sufficient to support a given level of parallelism is equal to the length of a combinatorial game. This game, which captures the difference between the centralized and the distributed problem, is PSPACE-Complete. This implies that unless NP=PSPACE, a scheduler cannot simultaneously minimize the communication cost and be computationally efficient.

The model presented can also serve as a framework for the study of distributed concurrency control by locking. For two transactions an efficient characterization of safe distributed locking policies is derived. The new graph-theoretic approach generalizes the geometric method used in the centralized case.

Thesis Supervisor: Christos H. Papadimitriou
Title: Associate Professor of Computer Science and Engineering

Key words: concurrency control, distributed database, communication, complexity, PSPACE-complete, games, locking.

Acknowledgements

I am particularly indebted to my thesis advisor, Professor Christos Papadimitriou, for his guidance and support. His suggestions of general research directions, his technical contributions, and his illuminating comments on the presentation of these results have been invaluable.

I would like to thank Professor Robert Gallager for his constant support and for inumerable discussions, which helped clarify and simplify many of the ideas in this thesis. I would like to thank Professor Peter Elias, for contributing his viewpoints to this thesis.

Many thanks are due to all my friends who have made the research environment at M.I.T. so exciting and my life as a graduate student so pleasant.

Most importantly I must express my gratitude to my parents, whose encouragement and warmth have always been my most valuable source of support.

This thesis was prepared with the support of National Science Foundation grants ECS-79-19880 and MCS-79-08965.

This empty page was substituted for a blank page in the original document.

Table of Contents

1. Introduction

There is now considerable literature, both theoretical and applied, concerning the **database concurrency control problem** - that is, maintaining the integrity of a database in the face of concurrent updates. Most of the theoretical work so far has been concerned with the **centralized** problem, in which the database resides at one site, and the update requests are submitted to a single process, called the scheduler, which implements the concurrency control policy of the database [7,25,37]. There is also some interesting applied work on **distributed** databases [2,3,4,28,36]. It is often said that the concurrency control problem is much trickier and harder in the distributed case, than in the centralized case. This is evidenced by the existing solutions, which are extremely complex and sometimes incorrect.

In this thesis we examine how the complexity of various problems, related to concurrency control, is affected when we attempt to solve them for distributed databases. The main focus is in two areas, serializability and safe locking policies, where efficient centralized solutions exist. Our approach and results also add to the theory of distributed computation, independently of their database context.

1.1 The Main Goals and New Results

Our main goal is to demonstrate the differences between the centralized and distributed versions of natural computational problems. We examine such problems from the area of database concurrency control, because we also wish to determine the limits of performance of concurrency control mechanisms.

We investigate two features of distributed computation, which distinguish it from centralized computation. First, the uncertainty of the order of events in a distributed environment [19]. The order of events is no longer best viewed as total, as in the centralized case; instead it is a partial order, whose structure depends on the number of sites of the distributed system. So our analysis will highlight differences between total and partial orders. The second element is the need for communication between sites, if the performance of an on-line distributed system is to match that of an on-line centralized system.

In order to find concrete differences we compare the computational complexity

of centralized and distributed tasks. We will use standard concepts from the theory of computational complexity, (i.e., deterministic polynomial time *P*, nondeterministic polynomial time *NP* or its complement *co-NP* and polynomial space *PSPACE*, [1,11,33,34]), as well as notions from the theory of combinatorial games [5,8,29]. The contributions of this thesis are summarized in the next three sections.

(I) The Model

We have developed a simple mathematical model of distributed databases, which captures the intricacies of distributed computation that are most pertinent to the database domain. Some novelties of our model are:

(1) User transactions are **arbitrary partial orders** of atomic steps, thus generalizing the straight-line programs of the centralized case. The order corresponds to both time-precedence and information flow, and it captures the notion of "distributed time".

(2) The scheduler, the concurrency control agent of the system, is itself a **distributed program**, consisting of communicating sequential processes [15], one for each site.

(3) **Redundancy** (the requirement that two entities stored at different sites be two copies of the same "virtual entity") is not treated at the syntactic level, but is considered as part of the integrity constraints of the database. Redundancy was at the root of the complexities of most previous attempts to formalize distributed databases.

As a consequence, there are **three measures of performance** in a distributed database (centralized theory deals with the first two):

(a) **Parallelism**, measured as the set of allowable interleavings of user actions.

(b) **Complexity** of the computational problems that the processes of the scheduler must solve.

(c) **Communication**, measured as the number of message exchanges between the processes of the scheduler.

A simple analysis, Theorems 1 and 2, verifies that the model is indeed a consistent generalization of the centralized model.

(II) Schedulers and Games

The three measures of performance of schedulers present interesting tradeoffs. For example, let us fix (a) (think of it as the parallelism specs of the system). By expending many messages, we can reduce the problem of distributed concurrency control to the centralized one (by broadcasting each request) and thus solve it in polynomial time for most reasonable specs [25]. It turns out that, based on a priori information about transactions, we can minimize the number of messages sent, by executing an exponential number of computation steps (and using polynomial space; this is the upper bound of our main result). Finally we cannot have a scheduler simultaneously using the minimum number of messages and running in polynomial time at each site, unless $NP = PSPACE$ (this follows from the lower bound).

Specifically our main result states that **for a certain parallelism specification (which in fact can be fixed to be the popular serializability principle** [3,17,25,31,40]) **minimizing communication costs is a computational problem complete for** $PSPACE$ [1,11,33,34]. Thus, our result appears to be concrete mathematical evidence suggesting that distributed concurrency control is indeed an inherently more complex task than centralized concurrency control (under quite general conditions, centralized schedulers can be implemented in polynomial time [25]).

Our result also adds to the literature on **distributed computation**, independently of its database context. It states, loosely speaking, that one cannot tell efficiently whether distributed processes can cooperate successfully for performing (an otherwise easy) **on-line** computational task, at fixed communication cost. It can therefore be considered as complementing the result of Ladner for lockout properties of "antagonistic" processes [18]. On the other hand, Yao has asked [38] whether minimizing communications costs for some distributed combinational computation is computationally intractable; we answer this in the case of an on-line computation.

The proofs of both our upper and lower bounds are quite intricate. For the upper bound we need a complicated characterization (Theorem 3) of the incomplete histories of actions (i.e., partial orders of events in the system) that can be completed

within a fixed number of messages. This upper bound holds for serializable histories, as well as for all similar parallelism specifications that can be achieved in a centralized manner. For the lower bound we relate distributed scheduling to a game played on graphs (the "conflict" graph of the transactions). Intuitively one player (Player II) is the distributed scheduler, and the other (Player I) is an adversary who submits user requests so as to force the scheduler to use as many messages as possible. Player I wants to prolong the game as much as possible, whereas Player II tries to bring it to an end as soon as possible (other than that there is no winner or looser). The rules are related in a simple way to the cycles of the graph. We prove that this game is complete for ***PSPACE***, and then show that our constructs can faithfully reflect a special kind of distributed concurrency control situation. Both steps involve intricate "gadget" construction (Theorem 4).

(III) Distributed Locking

A very common way of implementing concurrency control is by locking. In this method each entity is equipped with a binary semaphore (its lock) and transactions synchronize their operation by locking and unlocking the entities that they access. The purpose of locks is not mutual exclusion of shared resources as in operating system theory. Instead they are used to enforce correct sequencing of the indivisible transaction steps.

Locking policies have been extensively studied in the centralized case [7,13,21,26,30,39,40] and applied to distributed databases [22,23,35]. Our model provides a framework for the rigorous study of distributed locking.

The most elegant result in the theory of centralized locking is a geometric method, which efficiently characterizes the safe locking policies for two transactions. We examine the distributed version of this problem (i.e., when the transactions are partial orders instead of total orders of steps). We propose an alternative graph-theoretic approach for the centralized problem, which in addition provides an efficient sufficient condition for the distributed problem (Theorem 5). This condition is also necessary for transactions distributed at two sites (Theorem 6). Therefore this is a positive result (as opposed to the negative complexity results of Chapter 4). It also indicates how the difficulty of the problem may be affected by the number of sites at which we distribute it.

The material is organized as follows. Section 1.2 contains a review of database concurrency control, in which the various notions and results in the area are briefly described. Chapter 2 consists of the model definition (Section 2.1) and its simple properties, Theorems 1 and 2 (Section 2.2). The relation of distributed scheduling and games is rigorously established in Chapter 3. An upper bound on the complexity of the distributed problem is derived in Section 3.1 (Theorem 3). The games are defined in Section 3.2. Chapter 4 is an analysis of the complexity of these games and contains the main technical result, the lower bound in Section 4.1 (Theorem 4). The consequences of this result on the existence of schedulers are in Section 4.2. Chapter 5 provides a framework for the study of distributed locking (Section 5.1), and a characterization of safe two-transaction systems (Section 5.2), Theorem 5 for sufficiency and Theorem 6 for necessity. Finally, Chapter 6 contains the conclusions and a list of open problems and directions for further research.

The material on the model definition (Chapter 2) and distributed locking (Chapter 5) represents a joint effort with Prof. C.H. Papadimitriou. Part of this work, namely Chapters 2,3 and 4 appear in [16].

1.2 A Review of Database Concurrency Control

A database consists of a set of named data objects called *entities*. The values of these entities must at any time be related in some ways, prescribed by the *consistency requirements* (or *integrity constraints*) of the database. When a user accesses or updates a database, he may have to violate temporarily these consistency requirements, in order to restore them at some later time, with the specific data changed. For example, in a banking system, there may be no way to transfer funds from an account to another in a single atômic step, without temporarily violating the integrity constraint "the sum of all balances equals the total liability of the bank". For this reason, several steps of the interaction of the same user with the database are grouped into a *transaction*. Transactions are assumed to be *correct*, that is, they are guaranteed to preserve consistency when run in isolation from other transactions.

When many transactions access and update the same database concurrently, the consistency of the database may fail to be restored after all transactions have completed. If, for example, transaction 1 consists of the two steps

$x := x - 100$;

$x := x + 100$

and transaction 2 of the single step $x := 1.15 * x$, and the consistency requirement is simply "$x = 0$", then executing transaction 2 between the two steps of transaction 1 turns a consistent database into an inconsistent one. This is despite the fact that both transactions are *individually* correct, that is, each preserves database consistency when run alone. We must therefore find ways to prevent such undesirable interleaving, without excessively harming the average user delay and other measures of the efficiency of the system. This is the *database concurrency control* problem, already discussed extensively in the literature (see [37]).

In this section we present a brief (and by no means complete) review of the many results on concurrency control. We start by describing the elements of mathematical models used to study these problems in the centralized case. This setting will help us to present the theory of centralized database concurrency control (part-a). We then discuss how distributing the database affects the formulation of the problem and describe some of the proposed practical solutions (part-b).

(a) The centralized case

Intuitively a *database* consists of *entities* and a finite set of *transactions.* Each transaction is a total order on its *actions,* which are operations performed indivisibly. An action p of a transaction T is, in general, an *update* (i.e., a read and then a write) of an entity x_p, based only on the values of entities updated by actions that precede this action in the order of T.

A *history,* for a set of transactions $\boldsymbol{T}=\{T_1...T_m\}$, is a total order representing an interleaving of all transaction steps. It is therefore a total order respecting all transaction steps. It captures the order of events at the one site, where the database is stored. A *prefix* of a history h is an initial portion of h. H is the set of all histories, that is, all interleavings for *all* sets $\boldsymbol{T}$ of transactions.

We are interested in correct histories (i.e. histories that take the database from a correct initial to a correct final state). A well-known and generally accepted correct subset of H is that of *serializable* histories (SR). A *serial* history is one with no interleaving of actions of different transactions. A history is serializable iff it is equivalent (in the obvious schema-theoretic sense with uninterpreted function symbols for updates) to some serial history. Since each transaction is by itself correct a serializable history is obviously correct. Serializability has been widely recognized as the right notion of correctness (e.g., [2,3,4,17,25,31,40]). In fact it is shown in [17] that it is the most liberal notion of correctness possible, when only syntactic information (i.e., entity names) is available.

A *scheduler* is an algorithm handling incoming requests. It might use a priori information (e.g., the syntax of $\boldsymbol{T}$) and run time information (e.g., the order of incoming requests). The *input* and *output* of a scheduler are strings of actions in $\boldsymbol{T}$. In fact, one is the history of requests and the other the history of their execution. A scheduler is said to *realize a set of histories* C (where C is a subset of H) if:
(i) for all inputs, the output is a sequence in C,
(ii) for all inputs in C, the scheduler grants all requests immediately upon receipt. This captures the *on-line* and *optimistic* features of schedulers [25].

These sets C were proposed in [25] as a measure, whereby the performance of schedulers can be evaluated in a uniform setting. This measure expresses the class of all sequences of transaction steps that can be the response of the concurrency controller to a stream of execution requests. The richer this class, the fewer

unnecessary delays and rearrangements of steps will occur, and the greater the *parallelism* supported by the system.

A second measure of performance of a scheduler is the *computational complexity* of the decision problems it must solve.

The area of concurrency control was unified in [25] by formulating the problem as a relation between the two performance measures:

CC: The problem of **Concurrency Control** is, given a set C of correct histories, find a scheduler which realizes it and is computationally efficient.

A basic theorem in [25] is that such a scheduler exists iff the prefixes of C are polynomial time recognizable (i.e. in $\mathcal{P}$).

The obvious question in this setting is whether an efficient *serializer* (i.e., scheduler realizing SR) exists. The answer is yes. Testing a history for serializability, or a prefix for whether it has a serializable completion, is an easy task in the centralized case. The algorithm is based on *conflict graphs*. The conflict graph $G(\mathcal{T})$ for a transaction system $\mathcal{T}$ is a multigraph, with a node for each transaction in $\mathcal{T}$ and an edge between T_1 and T_2 labeled x, whenever T_1 and T_2 both update entity x. The order of executions of actions in a history assigns directions to the edges of $G(\mathcal{T})$. We call this *resolving the conflicts* between transactions. This result is the "folk" theorem of concurrency control [2,17,25,28,37]:

"A history h is serializable iff it resolves conflicts without creating directed cycles in $G(\mathcal{T})$. Similarly, a prefix has a serializable completion iff the already resolved conflicts do not create a directed cycle in $G(\mathcal{T})$."

The pioneering work in the field was [7], which also introduced concurrency control mechanisms such as *two phase locking* and *predicate locks*. It was followed by many interesting contributions (e.g. [2,13,31]). A number of concurrency control mechanisms were compared in the uniform setting of the parallelism measure C introduced by [25], where $C \subseteq SR$. Moreover it was shown, that if we distinguish between read and write actions then deciding whether a history is serializable (i.e. in

SR) becomes *NP-Complete* [25].

A very common way for implementing concurrency control is *locking*. In this method each entity is equipped with a binary semaphore (its lock) and transactions synchronize their operation by locking and unlocking the entities that they access. In fact, variants are possible in which locks of different kinds are defined, and certain kinds may coexist whereas others may not (e.g. shared or read locks, intention locks [13]). The lock-unlock steps are inserted in a transaction according to some *locking policy*. A locking policy may have the property that, if all transactions are locked according to it, then any execution respecting the locks is guaranteed to be serializable. Such a locking policy is called *safe*.

Given a transaction system T, there are certain well-known locking policies that can be applied to it. One is the *two-phase locking* (2PL) policy [7]. In it we insert locks surrounding the accesses of all entities, in each transaction subject to the following rule: The last entity to be locked is locked before the first entity is unlocked. Thus the transaction is divided into two phases: the *locking phase*, during which locks are acquired but not released, and the *unlocking phase*, in which locks are released but not requested. In an extremely conservative interpretation of this policy, we could lock all entities before the first step, and unlock them after the last. More reasonably, we could request locks for entities at the first step that they are accessed, and release locks at the end of the transaction. In fact, it is shown in [17] that the latter interpretation of 2PL is the best possible concurrency control, when syntactic information is acquired in an incremental, dynamic manner. It was first shown in [7] that 2PL is safe (though deadlock-prone).

If the entities are *unstructured* (that is, transactions access them in all possible patterns) then 2PL is the best possible locking policy. Suppose, however, that the entities form a tree, and are accessed by transactions as follows:

(i) A transaction accesses a subtree, whose root is the first entity to be accessed (after, of course, it is locked).

(ii) After this, when an entity is locked, its parent must be locked and not yet unlocked.

Then this locking policy, called the *tree* policy is shown in [30] to be both safe and deadlock-free. This holds for the more general *digraph policy* of [39]. In fact, the latter is generalized in [39] to the *hypergraph policy* which, it is proved, is the most general possible safe and deadlock-free policy.

Safe locking policies were characterized in [39]. The limitations of the parallelism that can be provided by locking were investigated in [26]. Safety of two-transaction locked systems can be efficiently decided [21], by employing a geometric methodology reminiscent of that used by Dijkstra for studying deadlocks [6]. Besides its independent interest and elegance, the two-transaction solution is the building block for resolving the general case. It turns out that a locking policy defined on d>2 transactions is safe iff all of its two-transaction subsystems are safe, plus a combinatorial condition. This combinatorial condition turns out to be ***NP-Complete***, but it is simple enough to have some interesting corollaries. For example, all specific locking policies mentioned above can be shown to be safe as immediate consequences of the condition.

(b) The distributed case

The assumption that the database is stored at one site is not always true. Distributing the database among various sites might be necessary and even desirable. In fact the current trend in technology is towards distributed databases [2,3,4,28,35,36].

In a distributed environment the transactions, histories and prefixes become partial orders and the scheduler consists of many communicating sequential processes, one at each site. The model presented in Chapter 2 abstracts the relevant properties of transactions, actions, histories, prefixes, and schedulers. It extends the parallelism measure of schedulers, the concept of serializability and conflict graphs to the distributed case. The new elements are, that the scheduler uses message passing between sites and that the conflicts are partitioned into the conflicts at every site. The problem of **Distributed Concurrency Control (DCC)** can be formalized as was that of Concurrency Control (CC). A rigorous treatment of this problem will require the selection of a formal system, in which to express distributed algorithms e.g. [9,15,24]. Such a system, with the least possible restrictions, is selected in the next chapter.

The problem of concurrency control has been examined by designers of distributed databases and various solutions have been proposed. Because of other important considerations in a distributed environment, concurrency control is viewed (and rightly so) as only one of a number of goals of such systems (e.g. other problems are, optimal partitioning of the database, distributed query processing [12],

properties of the communication medium, importance of deadlocks between sites [22,23], reliability of updates [14]). What is not clear from these involved distributed algorithms is, whether the distributed version of concurrency control, by itself, is a more complex task than its centralized version. This in fact is the subject of the present study.

A survey of distributed database concurrency control algorithms is contained in [4]. These algorithms are classified into methods using transaction timestamps to resolve conflicts [19] and methods using locking (particularly the two phase locking rule) [7]. The methods are compared on the basis of the three measures indicated in Section 1.1 (i.e. parallelism, complexity, communication), with an additional distinction between delaying or aborting requests that cannot be safely granted. Another issue that is investigated is the effect of having conflicts between read and write actions or write and write actions. There are methods, which cannot be classified into this timestamp v.s. locking scheme (e.g. voting methods used in [36]). There are also experimental comparative studies [10,20].

A concurrency control method, which stands out among all these algorithms is that employed by SDD-1 [2,3]. The reason for this is its preanalysis of a-priori information (i.e., the structure of the conflict graph) in order to enhance parallelism. An obvious question is, why should not a similar preanalysis be used to enhance the communication between the processes of the scheduler.

Finally let us mention a new research direction, which developed from the distributed problem, but is important even for the centralized case. It is tacitly assumed that there is one version of each entity in the database and an update creates a new version making the old one obsolete. It might be possible to use older versions in addition to the conflict graph, in order to perform concurrency control. This is done by changing the semantics of "read" and "write" (e.g., Reed's rule [27], before-and-after values [32]). This change in the model can have profound consequences, since it introduces a *space-parallelism* tradeoff (i.e., by using more versions the sets of interleavings C that can be realized by schedulers can be enriched).

2 A Model of Distributed Database Concurrency Control

This chapter contains the definition of our model for distributed database concurrency control. This model generalizes the centralized model, is simple and can be used for the analysis of all practical solutions proposed to date.

2.1 Model Definition

A distributed database is a collection of sites. Each site has its own processor and data. The sites are interconnected by a network and are controled by a distributed database management system (DDBMS). In Fig. 2.1 we show the architecture of a 2-site system; horizontal arrows join modules of the same distributed process. Formally, a distributed database is defined as follows:

Definition 1: A **Distributed Database Design** (DDD) is a quadruple $\langle G_D$, *Data, Stored-at, IC*$\rangle$ where:

(i) G_D=(V,E) is a graph, where every node corresponds to a site and every link to a two-way communication link between sites.

(ii) *Data* is a set of **variables** (or **entities**), denoted {x,y,z,...} (i.e. *physical* data items).

(iii) *Stored-at* : Data $\rightarrow$ V is a *function* that determines the site, where each physical data item is stored.

(iv) IC is a set of **integrity** constraints on the values of the Data.□

Note that multiple copies of the same *logical* data item are considered as different physical data items stored at different sites. The fact that they are copies and must remain identical for reasons of consistency is part of the integrity constraints, and is not treated separately.

The users interact with the database using transactions. In our model a transaction is a distributed program, not identified with a particular site.

Definition 2: A **transaction** T, in a given DDD, is a directed acyclic graph (dag) T=(N,A) such that:

(i) every node p is associated with one site of the system, *site(p)* and with an entity x_p stored at that site.

(ii) all nodes associated with the same site are totally ordered in A.

A **transaction system** $\mathcal{T}$ is a set of transactions $\{T_i\}$.□

Note that it is assumed that transactions are correct programs (e.g. update all copies of the same logical item in order to preserve the integrity of the database). We denote the partial order imposed by a transaction T_i on its actions as $>_{Ti}$.

Definition 3: The nodes of a transaction are the **actions** performed by the transaction. The semantics of an action p is the indivisible execution of the following two steps

$t_p := x_p$

$x_p := f_p(t_p, ..., t_q, ...)$ where q ranges over all actions that are ancestors of p in the transaction of p.

Here the t's are temporaries (i.e., a workspace local to the transaction) and the x's are physical items in the database. The f_i's are uninterpreted function symbols.□

Hence the nodes of transactions stand for indivisible actions. We do not specify the details of the exact nature of the computation performed by each action. Instead we view an action p of a transaction T as an uninterpreted function symbol f_p, with one output and $|\{q|\ q >_T p\}|+1$ inputs. The transactions are in fact program schemata, where all **updates** are treated by the concurrency control mechanism as uninterpreted updates. Designing the database (i.e., deciding how many copies of each item there are and where they are stored) and writing correct transactions (e.g., which copies to update, which other integrity constraints to satisfy) are problems at a higher level than concurrency control, and are not treated here.

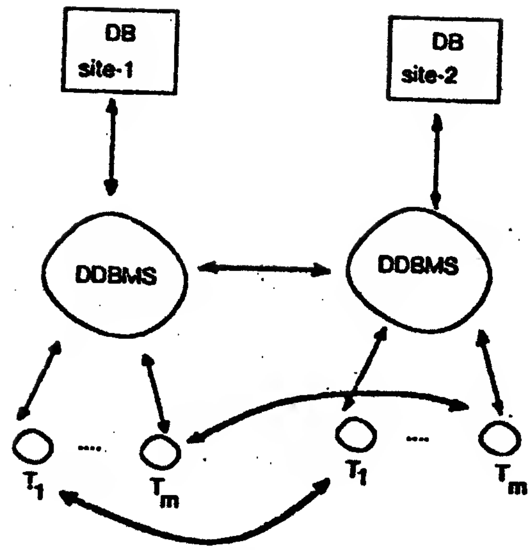


Figure 2.1 System Architecture

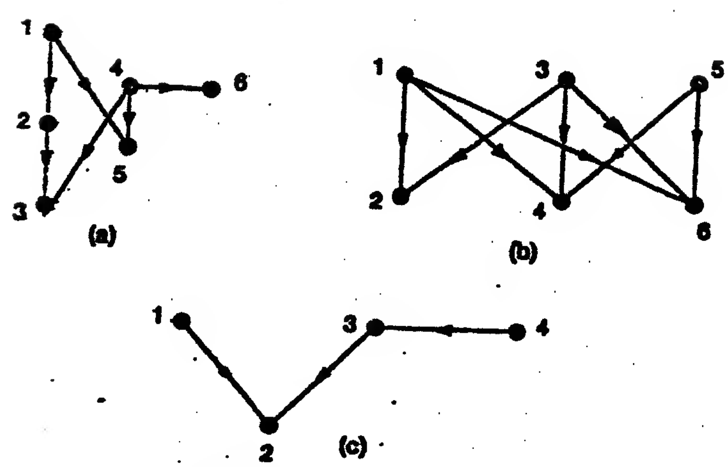


Figure 2.2 Transactions

The particular model of actions used was chosen for its clarity. Other models, such as those illustrated in examples 2 and 3 below could as well have been used, to produce results similar to those of Chapters 3 and 4.

Example 1: Consider the transaction of Fig. 2.2(a). Actions 1,2,3 are performed at site 1, actions 4,5 at site 2, and 6 at site 3. The actions performed at the same site are totally ordered. The actions are updates as in Definition 3, so every node can be associated with a variable and the site this variable is stored at. This model generalizes the centralized model of [17].

Example 2: Consider the transaction of Fig. 2.2(b) with actions (1,2),(3,4),(5,6) performed respectively at sites 1,2,3. If p is odd it is a read action with a *readset* of data items stored at its site. If it is even it is a write action with a *writeset* instead, and this update depends on all readsets (e.g., action 6 has writeset $W^6[x,y]$ and depends on readsets $R^1[w]$, $R^3[u,v]$, $R^5[x]$, where w is stored at 1, u,v at 2, and x,y at 3). This type of actions and transaction is used in SDD-1 [2,3].

Example 3: Consider the transaction of Fig. 2.2(c), where action j is performed at site j (there is only one action per site). *Dataset(j)*, of arbitrary cardinality, is updated based on its previous values and those of datasets of ancestor actions. This is a very simple model that makes the centralized version trivial (a transaction is an action), yet it presents interesting problems in the distributed case.

An edge in a transaction T between actions at different sites (called a *cross-edge*), denotes both temporal precedence and a transfer of information (i.e., in Fig. 2.2(a) update 5 needs data from update 1). These cross-edges correspond to user-defined messages, which the system must service.

A history is a description of a set of transactions and the process of their execution on the system. In a distributed system [19] it is in general impossible to tell which one of two events occured first, (because communication is not always instantaneous). Because of this uncertainty, we describe the execution order of the actions by a partial order. If two events are incomparable in this partial order, any one could have preceded the other. There are two restrictions on the partial orders. First, what happens at every site is totally ordered; this is consistent with the centralized problem and guarantees that the result of the execution is uniquely determined as in the case of individual transactions. Second, user-specified precedences are always respected. Formally:

Definition 4: A history is a pair $\langle T,\pi\rangle$, where $T=\{T_i, 1\leq i\leq m\}$ is a transaction system and π is a directed acyclic graph (dag) on the nodes of the transactions T_i such that:

(i) Nodes p with the same *site(p)* are totally ordered.

(ii) For any transaction T_i and actions $p,q\in T_i$ and $p >_{Ti} q$ we have that $p >_\pi q$ (where $>_\pi$ denotes the partial order imposed by π). □

Definition 5: A prefix of a history $h=\langle T,\pi\rangle$ is a pair $\langle T,\alpha\rangle$, where α is the induced subgraph of π by a subset of its nodes such that, if action $p\in\alpha$ all ancestors q of p in π belong to α.□

A history may be viewed as a special case of a parallel program schema (see Fig. 2.3). The resulting schemata and the rigorous treatment of their equivalence under Herbrand interpretation [25] closely resemble the centralized case.

Definition 6 :Two histories $h_1=\langle T,\pi_1\rangle$ and $h_2=\langle T,\pi_2\rangle$ are equivalent ($h_1\approx h_2$) iff their schemata are strongly equivalent (that is equivalent under the Herbrand interpretation of the function symbols and variables).□

Let H denote the set of all histories. Recall that a partial order can be considered as a set of total orders (those compatible with it). Let H^+ denote the set of all histories $\langle T,\pi\rangle$, where π is a *total order.* Therefore a history represents a particular subset of this basic set H^+. The histories with only transaction-defined cross-edges (arcs between actions at different sites) are maximal when considered as sets of total orders. Yet histories can have other cross-edges also (e.g.,arc (4,6) in Fig.2.3), whose presence restricts the allowable total interleavings of actions. The goal of concurrency control is to recognize on-line large sets of correct total interleavings.

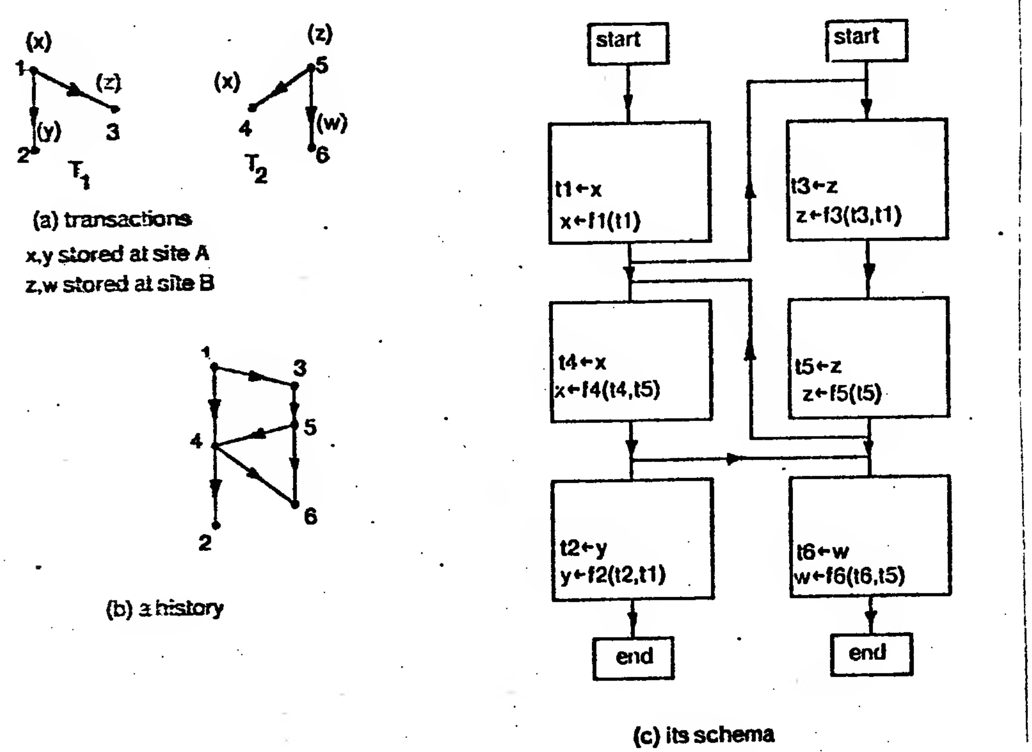

(x)
1
(z)
(y)
3
2
T_1
(z)
5
(x)
4
(w)
6
T_2
(a) transactions
x,y stored at site A
z,w stored at site B
1
3
5
4
6
2
(b) a history
start
start
t1←x
x←f1(t1)
t3←z
z←f3(t3,t1)
t4←x
x←f4(t4,t5)
t5←z
z←f5(t5)
t2←y
y←f2(t2,t1)
t6←w
w←f6(t6,t5)
end
end
(c) its schema

Figure 2.3

Since individual transactions are correct (i.e., take the database from a correct initial to a correct final state), histories in which transactions are executed one after the other (serial histories) are correct. Also those histories that are equivalent to them, called serializable, are correct. We denote the set of serializable histories by SR ($SR \subseteq H$).

Definition 7: A history h is serial iff

(i) The execution of actions at each site introduces a total order of transactions at that site (i.e. there are no transactions T_i, T_j $i \neq j$ with actions $p,q \in T_i$, $r \in T_j$ performed at the same site with p preceding r and r preceding q).

(ii) If T_i precedes T_j at one site it does so at all sites, where both transactions have actions.□

Definition 8: A history is **serializable** iff it is equivalent to a serial history.□

In the next section we will show that deciding serializability is an easy task. This task becomes ***NP-Complete*** if the model with read and write actions (instead of updates) is used [25]. Even in that case SR has interesting efficiently recognizable subsets (i.e., DSR[25]). What is significant, is that deciding whether a history is serializable in a centralized or distributed model are practically identical tasks. We discuss this similarity in the next section.

As in the centralized case, synchronization is necessary only between actions of a transaction system which operate on the same data (i.e., conflict). These conflicts are represented by the conflict graph $G(\mathcal{T})$.

Definition 9: For the transaction system $\mathcal{T}=\{T_i, 1\leq i\leq m\}$, the conflict graph $G(\mathcal{T})$ is an undirected multigraph (V,E), with a partial order $\geq_i$ associated to the edges incident upon each node i, such that:

(a) $V=\{i| \ 1\leq i\leq m\}$, with node i corresponding to transaction T_i.

(b) E is a multiset of edges. $E=\{$ copies of edge ij | for every copy of ij there is a distinct pair of actions $\{p,q\}$ with $p\in T_i$, $q\in T_j$, $i\neq j$ and $x_p=x_q\}$

(c) For two edges incident at node i we have ij $\geq_i$ ik iff the action in T_i corresponding to ij is identical to or precedes the action in T_i corresponding to ik.□

Note that an edge in E denotes a conflict between two transactions. Every edge ij in E corresponds to a pair of actions $\{p,q\}$, which update the same variable. Based on where this variable is stored we can partition E into as many multisets as there are sites (e.g., "red" and "green" edges for two sites). For an example see Fig. 2.4.

An **ordered mixed multigraph** $G=(V,E,A,\{\geq_i\})$ is a mixed multigraph with E a multiset of edges, A a multiset of directed edges and a partial order $\geq_i$ at each node i of the edges incident at the node. Conflict graphs are such objects with $A=\emptyset$.

Since a conflict (or an edge in $G(\mathcal{T})$) corresponds to two actions at the same site and a history $h=\langle\mathcal{T},\pi\rangle$ has a total order of the actions at each site, we can say that a history *resolves all conflicts*. That is, if edge ij corresponds to the pair of actions $\{p,q\}$, $p\in T_i$, $q\in T_j$, $i\neq j$, we direct ij from i to j iff $p>_\pi q$.

Definition 10: A prefix $\langle\mathcal{T},\alpha\rangle$ of a history assigns a direction (ij) to an edge ij of the conflict graph $G(\mathcal{T})$ iff *all* histories, which have $\langle\mathcal{T},\alpha\rangle$ as prefix, assign ij the direction (ij). Thus a prefix $\langle\mathcal{T},\alpha\rangle$ determines an **assignment of directions** to some edges of the conflict graph.

Conversely an assignment of directions to edges of the conflict graph is **realizable** by a prefix, if there is a prefix of a history assigning these directions and no others.□

Thus a prefix $\langle\mathcal{T},\alpha\rangle$ determines a unique ordered mixed multigraph $G^\alpha(\mathcal{T})$, which is $G(\mathcal{T})$ with some of its edges directed.

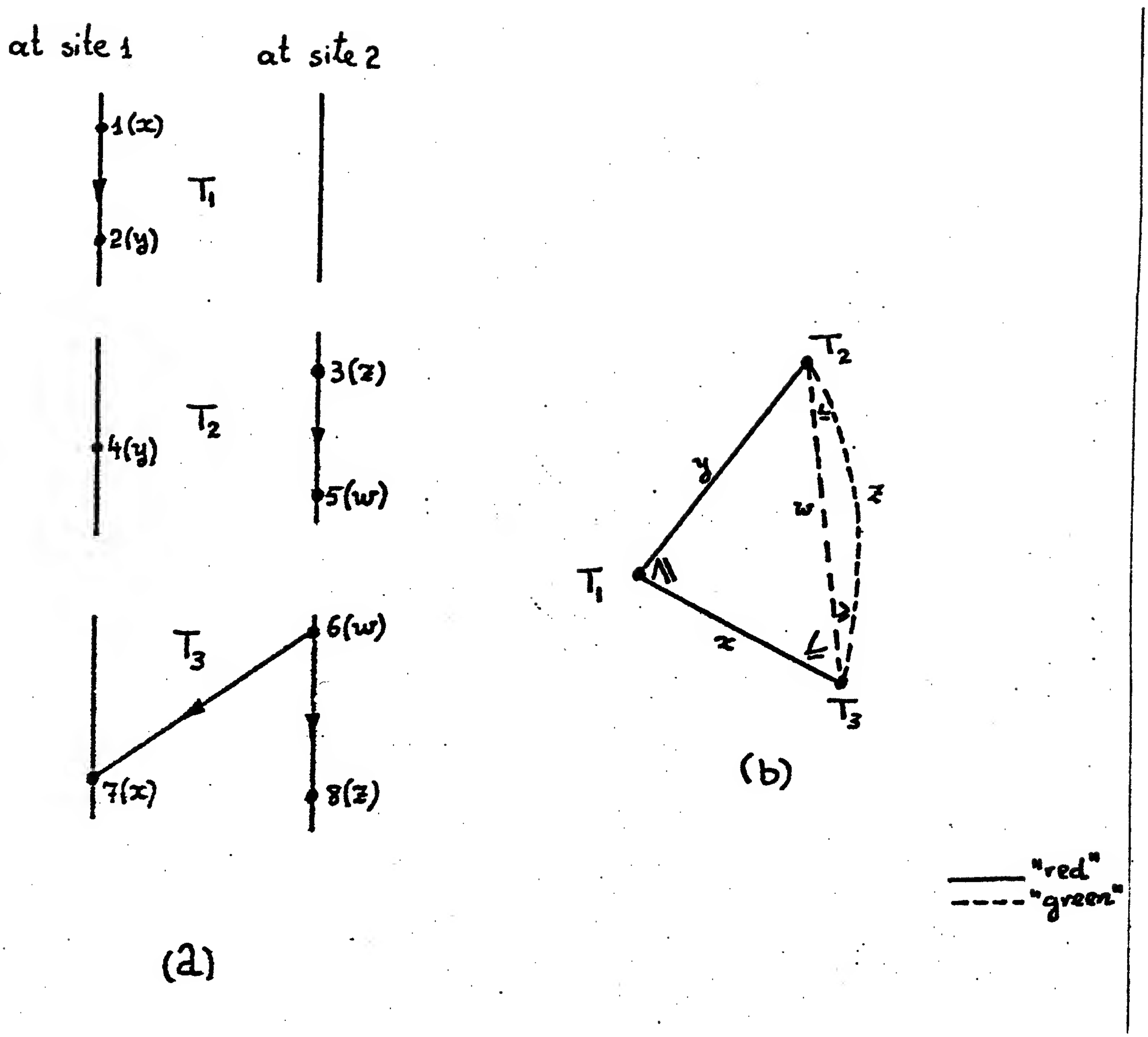


Figure 2.4
(a) Transactions (e.g. action 1 updates x)
(b) Conflict graph

Up until now the distributed problem appears to be a straight-forward generalization of the centralized case. What is considerably more complex in the distributed case is the subject of schedulers, and their design to meet performance specifications. For an exposition of the relatively simple theory for the centralized case see [25].

Our schedulers will be distributed algorithms characterized by the parallelism they provide and by their efficiency. We will measure parallelism using sets of histories C, that is subsets of H. The efficiency of the schedulers will be measured by the worst-case number of steps they execute and the worst-case number of messages they use. We will be interested in the following special C's:

<u>Definition 11</u>: Consider a set of histories $C \subseteq H$, such that for each $h \in C$ the only **cross-edges** (edges between actions at different sites) are defined by the transactions. Such a C we shall call a **concurrency control principle.**□

C is chosen in such a way, that all $h \in C$ are correct. The larger C is, the higher the level of parallelism supported by this concurrency control principle. Examples of concurrency control principles are serializability and serial (one-at-a-time) execution . Obviously, the former supports more parallelism. Thus concurrency control principles are very natural classes of histories measuring parallelism, although not all subsets of H can be expressed as such.

A **scheduler** Q is a *distributed algorithm.* (We do not explicitly specify the model of computation, although we shall use a concurrent language notation as needed). It consists of a set of communicating sequential processes [15], one for each site. Its instructions may involve the following:

1) Local Computation
2) Receiving an execution request for an action q.
3) Granting an execution request of an action q.
4) Sending a message to another site (i.e. *send*(<message>))
5) Receiving a message from another site

Each history h corresponds to a set $\{h^+\}$ of *total orders* (those that do not contradict h). Let h^+ denote any total order which respects the partial order of history h. If C is a set of histories, we let $C^+ = \{h^+, h \in C\}$. H is the set of all histories. An element of H^+ is a *string*, that is, a mapping from $\{1,2,...,n\}$ to N, where N is the set of all actions and $|N| = n$. In fact it is a pair $\langle T, string \rangle$, but we omit T when it is obvious from the context. The jth symbol of $h^+ \in H^+$ is denoted by h_j^+.

We thus assume that there is a total order on the arriving execution requests. This is a simplifying analytical tool (a formalism of the familiar notion of a timestamp) and is *not* used by the scheduler, whose processes still perceive the world in terms of partial orders. We therefore have a global *clock*, whose ticks are the arrivals of execution requests. This sequence of execution requests is the input of the scheduler. What is the output of a scheduler? It cannot be just a sequence of actions, as the relative ordering of the granting of requests with respect to their arrival is also important. The output of the scheduler is an n-tuple of strings $S = (s_1, s_2, ..., s_n) \in (N^*)^n$. Here s_j denotes the sequence of granted requests between the jth and (j+1)-st (after the jth if $j = n$) arrivals of requests. N^* is the set of all strings constructed from the set of actions N and includes the empty string. The concatenation of the n strings, conc(S), should be in H^+.

Thus a scheduler Q, besides being a distributed algorithm, is a nondeterministic mapping, (i.e. a set of mappings) from H^+ to $(N^*)^n$.

For each total order h^+, Q will produce a stream S of granted requests; one nondeterministic element is that of the various *communication delays*. A set of communication delays is a function d, which assigns to each execution of a *send* instruction by a process of Q a *nonnegative real number*. Not all functions are delay functions. The delay function has to be *feasible*, in that an action p must be executed before a successor q of p, in its transaction, can be requested. Note that the *zero function* $d=0$ is always a feasible delay function. Therefore the mapping $Q_d : H^+ \rightarrow (N^*)^n$ is well-defined for each feasible delay function d, assuming that local computation proceeds at a rate far faster than the arrival of requests and messages.□

Consider a set of histories $C \subseteq H$. Scheduler Q realizes C if all outputs of Q are in C- and thus presumably correct- and, furthermore, if Q is fed with a history in C and all delays are 0, then Q grants all requests without making them wait. It is argued in [25] that these are traits, in the centralized case, of all schedulers that are **on-line** and **optimistic** (two intuitive properties shared by all existing schedulers). The same arguments are applicable to justify Definition 12, where total orders and strings of actions are used to formalize this intuition.

Each process makes decisions about whether to grant or delay pending requests. These decisions can only depend on the **information** available to each process (i.e., *T* and the requests that it knows have been granted or are pending). This can be viewed as a consequence of the power of the set of instructions used (see above).

Definition 12: We say that Q is a **realization** of C iff
(a) $conc(Q_d(h^+)) \in C^+$ for all $h \in H$, and delay functions d.
(b) $Q_0(h^+)=(h_1{}^+,\ldots,h_n{}^+)$ for all $h \in C$.□

We illustrate the above definition in Fig.2.5. If $h^+ \in H^+$ is the input to Q there are many possible computation paths (i.e., sequences of events in the system). This is because of the essentially random delivery time of the messages. So every path has associated with it the delays of messages used along this path and has output $(s_1,s_2,\ldots,s_n) \in (N^*)^n$. The conditions are that the granted requests always form a correct history (a) and, moreover, if requested actions form a correct history and all delays are zero, then the requests must be granted immediately (b). These conditions must hold *for all* computation paths. So there is a difference between the use of the term nondeterminism above and that of classical complexity theory.

There also is a feedback effect from output to input (i.e., requests cannot be made if their ancestors in transactions have not been granted). This problem, which is due to our choice of an input-output description could restrict the set of inputs to a particular scheduler. Yet all prefixes of histories in C must still be inputs to all schedulers realizing C. This is also true for all prefixes not in C that are minimal (their prefixes are in C). These will be the only inputs of interest in Theorem 3.

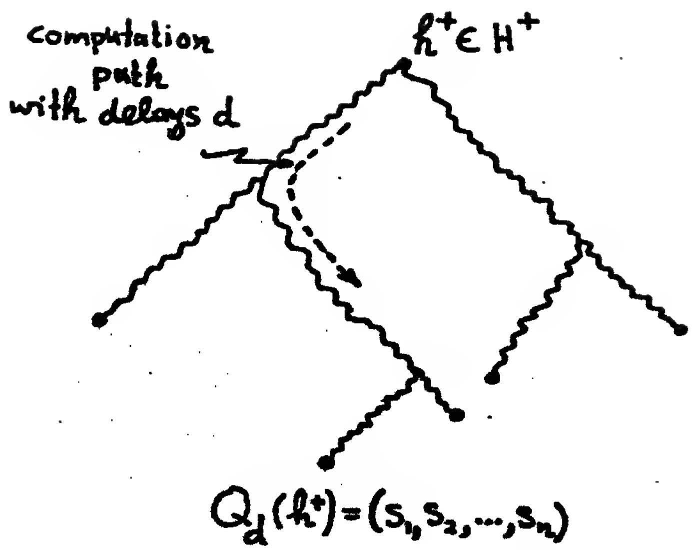


Figure 2.5

<u>Definition 13:</u> The **computational complexity** of Q is the worst-case sum of the counts of all local computations by Q over all processes of Q. The **communication complexity** of Q is the worst-case count of all *send* instructions executed by all processes of Q.□

Note that apart from the messages generated by the scheduler processes of the system there is also user defined communication, implied by transaction cross-edges (e.g. some action at site 2 needs data from site 1). *This communication is assumed free, since it is unavoidable, and can be used to pass information between scheduler processes at no cost.*

A scheduler Q is polynomial time bounded (or **computationally efficient**) if its computational complexity is bounded by a polynomial in n (i.e., n=|N|, N is the set of actions of *T*). This means that *all* possible computation paths have computational complexity (number of local steps) bounded by a polynomial in n.

We may even augment the computation power of our schedulers if we allow them, in their local computation steps, to consult an *oracle* [11] for a hard computational problem (say an ***NP-Complete*** problem). Many of our results will still hold for such schedulers.

Finally in order to characterize communication complexity we define the following classes $M_C(b)$:

<u>Definition 14:</u> For a prefix $\langle T,\alpha \rangle$ of C and an integer $b>0$ we say that : $\langle T,\alpha \rangle \in M_C(b)$ if there is a realization Q of C such that the total sum of *send* instructions executed at all processes of Q *after* $\langle T,\alpha \rangle$ is b or less.

Let $b^*(T)$ be the least b for which $\langle T,\emptyset \rangle \in M_C(b)$. A scheduler which achieves $b^*(T)$, for every T, is called **communication-optimal**.□

Note that $M_C(b)=\emptyset$ if $b<0$ and $M_C(b) \subseteq M_C(b+1)$. This definition describes the communication used if both processes of the scheduler are started with initial information $\langle T,\alpha \rangle$.

What Definition 14 says is that a priori information about the syntax of the transactions could be used to enhance the communication performance (worst-case number of messages used at run time) of the concurrency control mechanism. This is analogous to the conflict graph analysis used to improve parallelism in SDD-1 [2,3]. A communication optimal scheduler is the limit in message performance attainable, subject to a parallelism requirement C.

In Section 2.2 we will show that our model is a simple generalization of the centralized case and that there exists a computationally efficient scheduler realizing SR. In Chapter 3 we will recursively characterize the classes $M_C(b)$ and prove that there exists a communication optimal scheduler realizing SR. Finally in Chapter 4 we will examine the complexity of deciding whether a prefix is in $M_{SR}(b)$ and prove that, if *NP≠PSPACE*, no scheduler can realize SR and be both computationally efficient and communication optimal. This will be true even if we restrict our system to two sites, and our transactions to sequences of six updates each.

2.2 Properties and Limitations of the Model

The model presented in Section 2.1 consisted of extending the definitions of centralized concurrency control by introducing, where necessary, partial orders instead of total orders and by partitioning the conflicts according to sites. A more technical part was involved with defining the class of allowable distributed schedulers. We can now state the distributed problem we will examine:

<u>DCC</u>: The problem of **Distributed Concurrency Control** is, given a set of histories C (which we can prove correct), find a scheduler, which realizes C and is efficient (in terms of both local computation and communication).

Similarly to [25] we can prove:

<u>Theorem 1</u>: C has a computationally efficient realization iff the set of prefixes of C is in $\mathcal{P}$ (i.e., deterministic polynomial time).

<u>Proof</u>: Since we can expend an indefinite amount of communication between the different modules of a scheduler, the problem reduces to the centralized one (one site gathers all information and makes all decisions). Therefore the constructive proof of [25] is applicable. For arbitrary delays this construction gives us outputs in C^{+}; for 0 delays Definition 12(b) is also satisfied.□

Since the analysis we will be presenting deals primarily with the assignment of directions to edges of the conflict graph $G(T)$ by a prefix $\langle T,\alpha \rangle$, we need a characterization of realizable assignments (see Definition 10)

<u>Lemma 1</u>: Given a conflict graph $G(\mathcal{T})=(V,E,\emptyset,\{\geq_i\})$. An assignment of directions to a multiset X of its edges, producing the ordered mixed multigraph $(V,E\backslash X,A_X,\{\geq_i\})$ is realizable iff,

(a) If $ij \in X$ and is directed from i to j and $ik \geq_i ij$ then $ik \in X$.

(b) A_X has no directed cycles $(i_1 i_2 i_3 \dots i_n i_1)$ such that:

$i_1 i_2 \geq_{i2} i_2 i_3,\ i_2 i_3 \geq_{i3} i_3 i_4, \dots,\ i_n i_1 \geq_{i1} i_1 i_2$.

<u>Proof</u>: "only if" Given a prefix $\langle \mathcal{T},\alpha \rangle$ of a history let us first assign the direction (ij) to any edge ij in $G(\mathcal{T})$, which corresponds to a pair of conflicting actions {p,q}, under the following conditions:

(1) $p \in T_i,\ q \in T_j$

(2) $p \in \alpha$

(3) if $q \in \alpha$ then $p >_\alpha q$

Obviously all histories, which have $\langle \mathcal{T},\alpha \rangle$ as prefix resolve these conflicts in the same way. Moreover if an edge has not been given a direction then both its actions p',q' are not in α. We can complete $\langle \mathcal{T},\alpha \rangle$ with suffixes of histories that have p',q' in both orders. This proves that the directions we have constructed are exactly those assigned by $\langle \mathcal{T},\alpha \rangle$.

Because of causality both conditions (a) and (b) obviously hold for the directions constructed above.

"if" Given an assignment A_X we construct the following digraph (V_0,A_0)

V_0 (vertex set):

If $(ij) \in A_X$ and ij corresponds to conflicting actions {p,q}, $p \in T_i$ then $p \in V_0$.

If $p \in V_0$, $p \in T_i$ then all ancestors of p in T_i belong to V_0.

A_0 (arc set):

If p,q belong to the same T_i and $p >_{T_i} q$ then $(pq) \in A_0$.

If p,q correspond to an $(ij) \in A_X$ then $(pq) \in A_0$.

Since (b) is true (V_0,A_0) is acyclic and since (a) is true transaction precedences are respected. Thus (V_0,A_0) has the same nodes as some prefix and respects all its conflict resolving orderings (see "only if" part of the proof). By topologically sorting the nodes of (V_0,A_0) we can produce the desired prefix.□

We will now characterize the serializable histories and prove that the prefixes of SR are polynomially recognizable (in $\mathcal{P}$).

For the model of actions we are using (i.e., $t_p := x_p;\ x_p := f_p(t_p,...,t_q,...)$) we say that action p *reads a variable x from* q in history $h=\langle \mathcal{T},\pi \rangle$, if $x_p = x_q = x$ and q is the ancestor of p *closest to* p in π. The *reads x from* relation in our model is always a chain of all actions p, for which $x_p = x$. The chains for all x's give us the *reads-from* relation. It is easy to see that we can represent the *reads-from* relation for a given history $h=\langle \mathcal{T},\pi \rangle$ as a directed multigraph D(h), with nodes corresponding to transactions and edges corresponding to edges of these chains (labelled by the variable read and the action reading it). In D(h) we can ignore arcs of the form (i,i) because we can deduce these from $\mathcal{T}$.

Since histories are program schemata, we have from standard schemata equivalence theory [25]:

Proposition 1: Two histories $h_1=\langle \mathcal{T},\pi_1 \rangle$ and $h_2=\langle \mathcal{T},\pi_2 \rangle$ are equivalent iff $D(h_1)=D(h_2)$ (i.e, they have the same actions and the same *reads-from* relation).□

For other models of actions it is necessary to distinguish between live and dead transactions [25]. In our model, all transactions are live. Obviously for a serial history h_s, $D(h_s)$ is acyclic.

The following theorem (an obvious generalization of the centralized case) is yet another variant of a veritable "folk" theorem [3,17,25,28,40]:

Theorem 2: A history h is serializable iff it resolves conflicts without creating directed cycles in $G(\mathcal{T})$. Similarly, a prefix has a serializable completion iff the already resolved conflicts do not create a directed cycle in $G(\mathcal{T})$.

Proof: Let D(h) represent the *reads-from* relation for h. If $h \approx h_s$ for h_s serial then $D(h)=D(h_s)$, which is acyclic. If D(h) is acyclic we can find a total order of transactions by topologically sorting it and then consider the serial history which

respects this total order on all processors. This serial history has the same D(h). The only difference from the centralized case is that D(h) can be partitioned into as many subdags as there are sites.

It is easy to see that D(h) is acyclic iff $G(T)$ with the assigned directions is acyclic. A scheduler, recognizing serializable interleavings and knowing of all requests (operating in a centralized manner), would arbitrate requests on-line by making sure that the assignment of directions to the conflict graph introduces no directed cycles. This can be done in polynomial time. Therefore there is a computationally efficient scheduler realizing SR.□

It is easily seen from the above analysis that histories with the same total orders on each site are equivalent, and cross-edges are not needed for deciding serializability. These edges, between actions at different sites, can be used in relating histories and performance of distributed schedulers.

Let us end this Chapter with a brief discussion on the properties of our model. The advantages of this model are:

(a) generality: All models of transactions and schedulers proposed have the properties of our model. Variations in the format of transactions (i.e. defining separate read and write actions) do not affect the results that will be presented.

(b) mathematical simplicitly: All cases are treated uniformly (i.e. copy equivalence is just one more instance of the integrity constraints). All questions are reduced to questions on concrete combinatorial objects (e.g. conflict graphs). There are no hidden assumptions since the performance measures (parallelism, computation steps, messages) and the model of distributed algorithms are well-defined.

(c) compatibility: The model is an extension of the centralized case. In Section 5.1 we will be able to express distributed locking policies in the model, just as was done in the centralized case.

(d) correctness: Serializability is not the only notion of correctness, but it is certainly the most generally accepted one. It is intimately related to the a priori information about the syntax of T.

On the other hand there are some disadvantages:

(e) Restricting attention to the three measures of performance: We ignore goals which are important for distibuted systems but hard to treat mathematically (e.g. reliability of the update mechanism, which is usually handled by *two phase commit* protocols[14]).

(f) The assumption that all syntactic information is known at run time: Information about transactions is not always available before the transaction is initiated. There is a whole spectrum of possibilities, between total syntactic information being known before run time (static case) and the completely dynamic case, in which information is acquired for each action separately as it is presented for execution.

(g) The measure of parallelism used (i.e., the size of the set $C \subseteq H$) is a crude approximation of the average user delay [25].

These disadvantages are shared by most formal work on database concurrency control.

3. Communication-Optimal Schedulers and Games

We will now state and prove a theorem, which relates the structure of histories and their prefixes with the number of messages necessary and sufficient to achieve a performance C.

3.1 A Recursive Characterization of Communication Complexity

As defined in Section 2.1 the performance measure for parallelism C is a set of histories (i.e. $C \subseteq H$). In this section we require C to be a *concurrency control principle* (see Definition 11). Concurrency control principles are very natural classes of histories measuring parallelism (examples are serializability SR, and serial execution S). Let PR(C) be the set of prefixes of histories in C. Two properties of C are used in our recursive characterization of communication complexity. First, if C is a concurrency control principle, then for each $h \in C$ the only *cross-edges* (edges between actions at different sites) are defined by the transactions. Second, we have an efficient (polynomial time in n) test of membership of a prefix in PR(C) (for example, if C=SR Theorem 2 provides us with such a test). If no such test is possible, concurrency control is quite hopeless, even in the centralized case [25].

Let us briefly review the notation used. A prefix is denoted as a pair $\langle T,\alpha \rangle$, where T represents the transactions (a priori syntactic information) and α the order in which some actions were executed. We use α for $\langle T,\alpha \rangle$ when there is no ambiguity about T. Also $(\beta/\alpha)_i$ denotes the prefix of β that contains α and all actions of β at site i (the *projection* of β at site i given α). So α is a prefix of β and $(\beta/\alpha)_i$. Finally we use $M_C(b)$, where $M_C(b) \subseteq$ PR(C), for the set of all prefixes $\langle T,\alpha \rangle$ of C such that there is a realization of C which, when started with $\langle T,\alpha \rangle$, *sends* b or fewer messages.

<u>Theorem 3</u>: Let C be a concurrency control principle, $\langle T,\alpha\rangle$ a prefix in PR(C), and b a nonegative integer. Let i denote an index ranging over the site number $i\in\{1,2\}$. Then the following are equivalent:

(I) $\langle T,\alpha\rangle \in M_C(b)$

$$\text{(II)}\ \forall \langle T,\beta\rangle \ \text{ if} \begin{cases} (1)\ \langle T,\beta\rangle \notin PR(C) \\ (2)\ \forall i\ \langle T,(\beta/\alpha)_i\rangle \in PR(C) \end{cases} \text{then} \begin{cases} (3)\ \forall i\ \langle T,(\beta/\alpha)_i\rangle \in M_C(b) \\ (4)\ \exists i\ \langle T,(\beta/\alpha)_i\rangle \in M_C(b\text{-}2) \end{cases} \square$$

Less formally (II) reads as follows:
"For all continuations α_1,α_2 of α such that α_1 is α with some actions at site 1 added, and α_2 is α with some actions at site 2 added, and such that their least common continuation β is not a prefix of C (while α_1,α_2 are) the following holds: $\langle T,\alpha_1\rangle,\langle T,\alpha_2\rangle\in M_C(b)$ and one of them is in $M_C(b\text{-}2)$."

We will first give an intuitive interpretation of Theorem 3 (which is illustrated in Fig. 3.1). Consider a scheduler, which realizes C, starts from $\langle T,\alpha\rangle$ and receives input requests $\langle T,\beta\rangle$. Each one of the scheduler processes i, $i\in\{1,2\}$, can see $(\beta/\alpha)_i$, without sending any messages. This is because process i (e.g. process 1 in Fig. 3.1), knows α (e.g. $\emptyset$ in Fig. 3.1), receives the actions of β to be executed at site i (e.g. actions 4 and 5 in Fig. 3.1) and using the transaction-defined messages (e.g. action 5 needs data from action 6 in Fig. 3.1) can learn about some actions at the other site (e.g. actions 6 and 8 in Fig. 3.1).

A situation that forces communication is one where the projections of the input, that each process sees directly, seem correct (i.e. $\langle T,(\beta/\alpha)_i\rangle \in$ PR(C)) and therefore must be executed on-line to achieve the goal C, yet the real input could be incorrect (i.e. $\langle T,\beta\rangle \notin$ PR(C)). For the example in Fig. 3.1, $\alpha=\emptyset$ and there is a unique minimal "bad" continuation β. We use α_i as a shorthand for $(\beta/\alpha)_i$, when there is no ambiguity.

Theorem 3 tells us that these are the only cases for which we need communication between scheduler processes; furthermore to guard against such "bad" β's only one $(\beta/\alpha)_i$ (say $(\beta/\alpha)_{i*}$ or α_{i*} for short) has to be in $M_C(b\text{-}2)$. The communication protocol is built in such a way, that the corresponding site i^* will ask

for the approval of the other site in order to execute α_{i*}. There is therefore a balancing of the *send* instructions among the two processes of the scheduler, with each *send* instruction guarding against a "bad" β.

The rigorous proof of Theorem 3 is given below. In one direction it entails an adversary argument and case analysis. For the other direction we give an explicit recursive construction of scheduler processes that realize C, within the prescribed number of messages. The basic idea of this construction is the following: Let α_1,α_2 be correct continuations of α and projections of an incorrect β. Let Q_i $(i=1,2)$ be a message-optimal protocol, given that α_i has been executed. Then the Q_i's can be combined to produce a Q that is message optimal, given that α has been executed. If Q_i uses more messages than Q_j, then the process of Q at site j will have the *send* instruction guarding against β.

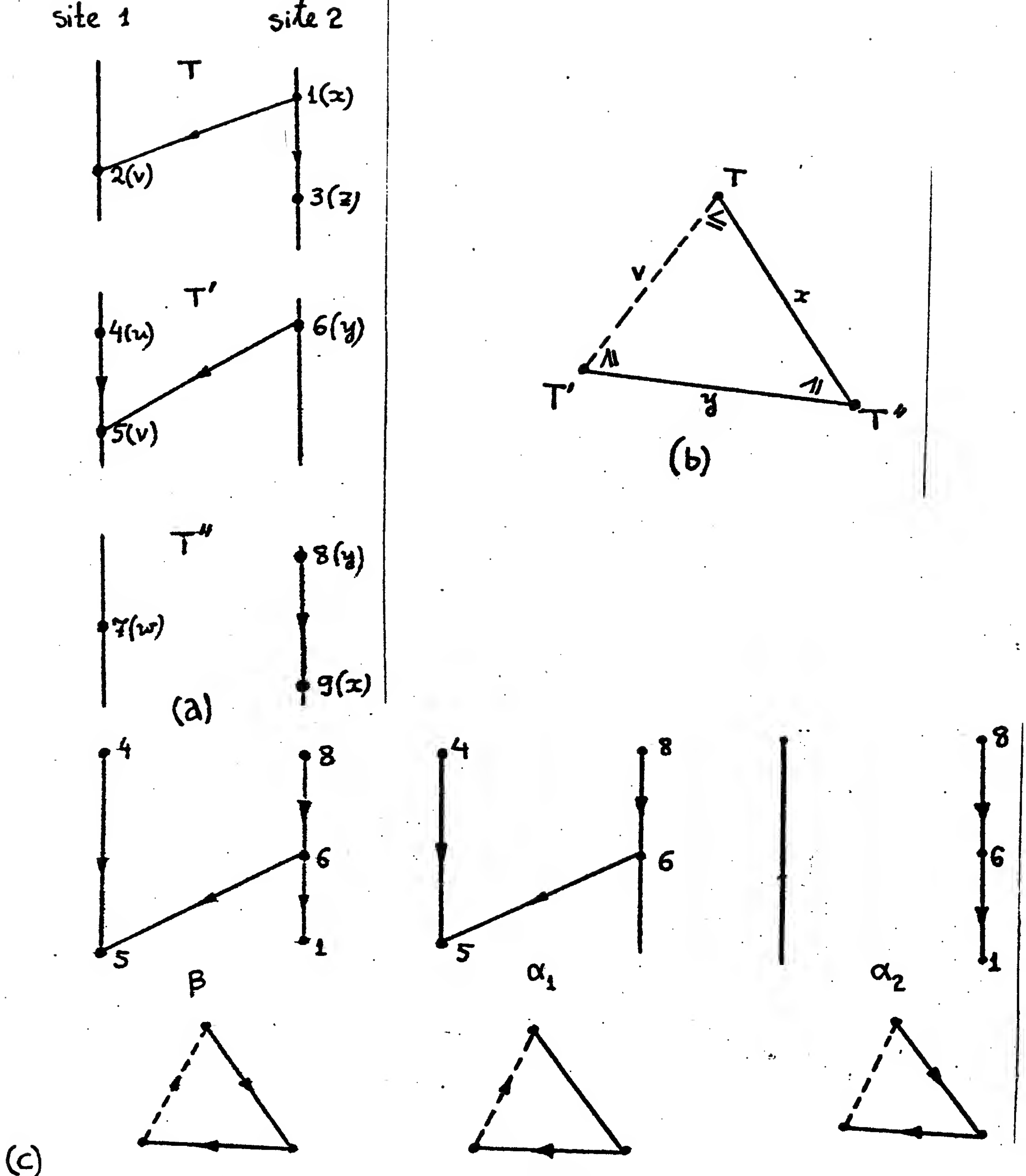


Figure 3.1

(a) Transaction system (u,v,w at site 1, x,y,z at site 2) (e.g. action 1 updates x)

(b) Conflict graph (i.e. ---- =conflicts at site 1, —— = conflicts at site 2)

(c) Illustrating Theorem 3. Above: prefixes. Below: assignments of directions.

Proof of Theorem: Let α_i denote $(\beta/\alpha)_i$. Theorem 3 recursively characterizes $\langle T,\alpha\rangle \in M_C(b)$, based on prefixes $\langle T,\alpha_1\rangle$, $\langle T,\alpha_2\rangle$, which properly contain $\langle T,\alpha\rangle$. The containement is proper because of conditions (1),(2) of the Theorem. The last actions of $\langle T,\beta\rangle$ at sites 1 and 2 (p_1 and p_2 respectively) are concurrent and not contained in α. Consequently α_1 containing p_1 and α_2 containing p_2 are not prefixes of each other. Note that in order to terminate the recursion we use the following facts: if $b<0$ then $M_C(b)=\emptyset$ and if h is a history in C then $h\in M_C(0)$. For $b=0$ the statement of the theorem becomes: "$\langle T,\alpha\rangle \in M_C(0)$ iff no prefixes $\langle T,\beta\rangle$ exist satisfying conditions (1),(2) and (3)".

"I⇒II" We will now prove that if β exists with properties (1),(2) and (3) and $\{\langle T,\alpha_1\rangle \notin M_C(b)\} \vee \{\langle T,\alpha_2\rangle \notin M_C(b)\} \vee \{\langle T,\alpha_1\rangle \notin M_C(b\text{-}2) \wedge \langle T,\alpha_2\rangle \notin M_C(b\text{-}2)\}$ then $\langle T,\alpha\rangle \notin M_C(b)$. This is obvious if one of the two first members of the above or clause is true. If both are false but the third member is true we will prove that communication involving *two* messages is forced, between the execution of $\langle T,\alpha\rangle$ and that of $\langle T,\alpha_1\rangle$ or $\langle T,\alpha_2\rangle$ for all schedulers realizing C. For this we will use the general specifications for a programming system as outlined in Section 2.1.

Consider the following situation that the process of the scheduler at site 1 (site 2) can face. It receives request $p_1(p_2)$, while knowing that certain requests $\langle T,\gamma\rangle$ have been granted with $\gamma=\alpha_1\text{-}\{p_1\}$ ($\gamma=\alpha_2\text{-}\{p_2\}$). It has to decide whether to grant or delay $p_1(p_2)$. If it grants the request, then according to its local view of the input the result would be correct. Its local view of the input can be the actual input, that is it could be the case that the input history is in C, it has $\langle T,\alpha_1\rangle$ ($\langle T,\alpha_2\rangle$) as a prefix, and no other requests have been submitted at the other site yet. Therefore the scheduler cannot delay $p_1(p_2)$ for the purpose of waiting for some future request submitted at site 1 (site 2). It has the following two options. First, the process of the scheduler at site 1 (site 2) can either grant p_1 (p_2) directly or after receiving a message from the process at the other site. Second, it can inform the other site of $p_1(p_2)$ or it can withhold that information. These two options expressed as sets of instructions in our programming system give rise to the only four possible cases for site 1 (site 2) to handle $p_1(p_2)$. These are cases A1-A4 (cases B1-B4 are symmetric).

Case A1: if (input as seen at site 1 is in PR(C)) then grant p_1

In this case the process at site 1 does not wait or inform site 2 of its decision.

Case A2: if (input as seen at site 1 is in PR(C)) then grant p_1
send (message to site 2)

In this case the process at site 1 does not wait but informs site 2 of its decision. The message can potentially contain all available information at site 1. The order of these instructions can be interchanged.

Case A3: wait (for message from site 2)
if (input as seen at site 1 is in PR(C)) then grant p_1

In this case the process at site 1 waits for information from site 2, but does not send any information. Interchanging the order of these steps will be treated similarly with case A1.

Case A4: send (message to site 2)
wait (for message from site 2)
if (input as seen at site 1 is in PR(C)) then grant p_1

In this case the process at site 1 informs site 2 of its problem, and waits for an answer before proceeding. Any permutation of these steps also uses two messages.

We will now reach a contradiction by examining two possibilities.

(i) If either the process at site 1 uses the instructions of case A4 or the process at site 2 uses the instructions of case B4 then two messages are consumed in executing either $\langle T,\alpha_1\rangle$ or $\langle T,\alpha_2\rangle$. Since we assume these prefixes belong to M_C(b) and not to M_C(b-2) and they are prefixes of $\langle T,\alpha\rangle$, we will have to use (b-1)+2=b+1>b messages at least to achieve our performance goals starting from $\langle T,\alpha\rangle$.

(ii) For all other combinations of cases of instructions we will also find contradictions.

Using case Ai instructions for site 1 and case Bj instructions for site 2, for i,j$\in$ {1,2}, we obviously have situations where the input prefix is $\langle T,\beta\rangle \notin$PR(C) and

is (incorrectly) executed without rearranging requests.

In any one of the remaining combinations either site 1 uses instructions of case A3 or site 2 uses instructions of case B3. We will reach a contradiction using A3 instructions (B3 is symmetric). Let the input history h^* be in C and have $\langle T,\alpha_1\rangle$ as prefix. When the request for p_1 will be submitted to the process of the scheduler at site 1, the process will wait for a message from the other site, which will determine its decision (granting p_1 or making it wait for future requests from other transactions). But when actions of $\langle T,\alpha_1\rangle$ are being executed at site 2 no such message can be sent. This is because according to site 2 both $\langle T,\alpha_1\rangle$ and $\langle T,\alpha_2\rangle$ are possible (proper) continuations and decisions cannot be made excluding one or the other. So the message site 1 is waiting for will be sent when descendants of $\langle T,\alpha_1\rangle$ arrive at site 2. Thus we force action p_1 to wait for some action which is not its ancestor in h^*, and therefore h^*, although in C, is not executed on-line as required by Definition 12 of Section 2.1.

"II⇒I": Under the conditions of the theorem we will construct a realization of C achieving the desired performance. That is we will present a scheduler, which will consist of two processes (i.e. **LOCALSCHED**$_i(\langle T,\alpha\rangle,b)$, $i=1,2$) and recognize on-line all histories in C with $\langle T,\alpha\rangle$ as prefix, without executing more than **b** *send* instructions in the worst case. The algorithm is written in a programming system with the capabilities outlined in Section 2.1.

The **LOCALSCHED** processes (see Fig. 3.2 for $i=1$) communicate with transactions and with each other using *messages*. The messages received by a process are buffered in a FIFO queue. The variables that the scheduler processes use for recording the state of the system are the *state variables* s_i, r_i, t_i, p_i, and b. The variables m_i (modes) are used to synchronize the two processes, so that when one process asks the other a question it expects an answer before examining other requests. The execution of *send* instructions is controled by the conditions of Theorem 3. The procedures **Grant**$_i$ grant requests. Finally the procedures **Delay**$_i$, **Delay***$_i$ handle the cases where the input is discovered to be incorrect. Let us explain the above features in some detail.

$LOCALSCHED_1(\langle T,\alpha\rangle,b)$

1. $s_1:=\langle T,\alpha\rangle$; $r_1:=\langle T,\alpha\rangle$; $t_1:=\emptyset$; $p_1:=\emptyset$; $m_1:=$*normal*;
2. when queue nonempty do
3. if $m_1=$*normal*
   then M:=first message of queue (delete it from queue);
   else wait (for message of type Q or A); M:=first such message;
4. (Based on M assign)
   s_1:=(state of 1);
   r_1:=(state of 1 that is also known by 2);
   p_1:=(set of pending requests, at most one per site);
   t_1:=(state of 1 resulting if pending requests were granted);
5. (Respond to message M) do one of three cases (R,A,Q);
6. **od end**

case R:

if $t_1 \notin PR(C)$ **then** $Delay_1(p_1)$ **else**

if $\exists\beta$ s.t. $\{t_1=(\beta/r_1)_1\} \wedge \{\beta \notin PR(C)\} \wedge \{(\beta/r_1)_2 \in PR(C)\} \wedge \{t_1 \in M_c(b\text{-}2)\}$

then $m_1=$*wait*; **send** $\langle 2,Q,p_1,s_1\rangle$;

else $s_1:=t_1$; $Grant_1(p_1,s_1)$;

case A:

if p_1 is in s_1 **then** $Grant_1(p_1,s_1)$; $LOCALSCHED_1(s_1,b\text{-}2)$; **else** $Delay^*_1(p_1,s_1)$;

case Q:

if $t_1 \in PR(C)$ **then** $s_1:=t_1$;

if $m_1=$*normal* **then send** $\langle 2,A,\emptyset,s_1\rangle$;

if $t_1 \in PR(C)$ **then** $Grant_1(p_1,s_1)$; $LOCALSCHED_1(s_1,b\text{-}2)$; **else** $Delay^*_1(p_1,s_1)$;

Figure 3.2 LOCALSCHED at 1

(a) Messages: The messages received by the scheduler process at site 1 (for those received at site 2 interchange 1 and 2) have the following format, (i.e. there are three types of messages): <1, type, requested action, state at site 2>.

R (for type=request). This is a message from a transaction to the scheduler process at site 1. It contains a request for an action p at site 1. State information about site 2 is included (else it is ø), when data from site 2 is necessary to compute p. This happens when an ancestor of p, in the transaction of p, has been executed at site 2. Then the transaction defined message can be used to transmit information about the state at 2. Examples of such messages are $\langle 1,R,p,s_2\rangle$ or $\langle 1,R,p,\emptyset\rangle$.

Q (for type=question). This is a message from the scheduler process at site 2. This process needs site 1's approval in order to decide whether to grant some request p, when it is at state s_2. An example for such a message is $\langle 1,Q,p,s_2\rangle$.

A (for type=answer). This is a message from the scheduler process at site 2 answering a type Q message of the process at site 1. Site 2, having full knowledge of the system, determined whether the pending request at site 1 should be granted. All necessary information has been incorporated in the state at 2. An example for such a message is $\langle 1,A,\emptyset,s_2\rangle$.

(b) State: The state of each **LOCALSCHED**$_i$ (s_i) is the prefix in PR(C), that the process at site i knows has been executed. For example with C=SR the state is $G(\mathcal{T})$ (see Definition 9 Section 2.1), with a partial assignment of directions that can be realized by a prefix. For this case correctness is guaranteed if acyclicity is maintained in the directed part of the conflict graph. In addition to s_i **LOCALSCHED**$_i$ keeps an estimate of the state of the other scheduler process r_i. With this estimate it keeps track of the part of s_i that the other site might not have heard of. Every time a message is received or a request is granted s_i and r_i are consistently updated. Finally p_i is used to store pending requests and t_i the state that would result if these requests were granted. The variable b keeps count of site i's estimate of the number of *send* instructions executed or the number of messages of types Q and A.

(c) Synchronization of the scheduler processes: The modes (m_i) are binary variables used by the scheduler processes to guarantee that every question is answered. A mode is either *normal*, indicating that new requests are processed, or

wait, indicating that the process at i needs an answer in order to decide on pending requests and handles no requests until it receives one. As can be seen from Fig. 3.3 a type A message is never received when the mode is *normal.* The two sites never deadlock (wait for each other indefinitely), because of the effect of A and Q type messages on the mode.

(d) Communication Protocol: Every incoming request is examined (if the mode is normal) and if it renders the local state incorrect it is delayed. If its execution leads to a correct local state (t_i) we determine whether *send* instructions should be executed. We first examine whether it is possible for a malicious adversary to give as input to the other site requests, also leading to a correct local state for the other site, but such that the total input is incorrect. If this is not possible the request is granted. If, on the other hand, this is possible some strategy has to be worked out for communication. In that case we also test whether $t_i \in M_C(b\text{-}2)$. If this is not so the request is granted without informing the other site. If this is so, site i *sends* a Q message in order to ask for the other site's permission to proceed. If it receives a go-ahead then we notice that, after sending two messages, both local processes are in fact **LOCALSCHED**$_i(s_{new}, b_{new})$ with common new state and new message parameters. This makes it possible to give an inductive proof of correctness.

Three decision questions are actually answered:

$\{t_i \in PR(C)\}$?

{does a "bad" β exist with t_i=(projection of β at i given r_i)}?

$\{t_i \in M_C(b\text{-}2)\}$?

(e) Granting requests: When **LOCALSCHED**$_i$ decides to grant a request it allows the transaction to update the variable of the requested action. Also if this transaction will send a message to some other site it will incorporate in that message the local state s_i. All this is achieved using **Grant**$_i(p_i,s_i)$ (i.e., if p_i contains a request for an action at site i, then let the transaction of this action perform its update and use s_i in any messages it sends to the other site, else no operation).

(f) Delaying requests: If a request is received when the mode is *wait* the request remains in the queue and will eventually be processed in its order of arrival. It is delayed at most by the communication delay of a Q and an A message. If on the other hand the scheduler discovers that the pending requests (at most one at each site) would lead to an incorrect execution then it delays one pending request. There

are two cases:

For only one site $t_i \notin PR(C)$. Then the process at i delays the pending request at i by putting it at the end of its queue (busy waiting). The scheduler continues functioning as if the input were correct. This is accomplished using $Delay_i(p_i)$ (i.e., if p_i contains a pending request at i, then put it at end of i's queue, else no operation).

Both sites discover that $t_i \notin PR(C)$. This happens through an exchange of a Q and an A message (one pending request at the site that sent the Q message) or of two Q messages (one pending request at each site). In this case $Delay^*_i(p_i,s_i)$ is used. One pending request is delayed. If there are two pending requests the younger one is delayed and the older one granted. Since consistent timestamps [19] can always be assigned to events in a distributed system, there is no problem in determining the younger of the two pending requests. Both processes of the scheduler know that the input is incorrect and that a common prefix s* has been executed. In this case no more *send* instructions have to be executed to realize C (see Def. 12 Section 2.1), because a predetermined correct completion of s* can be executed.

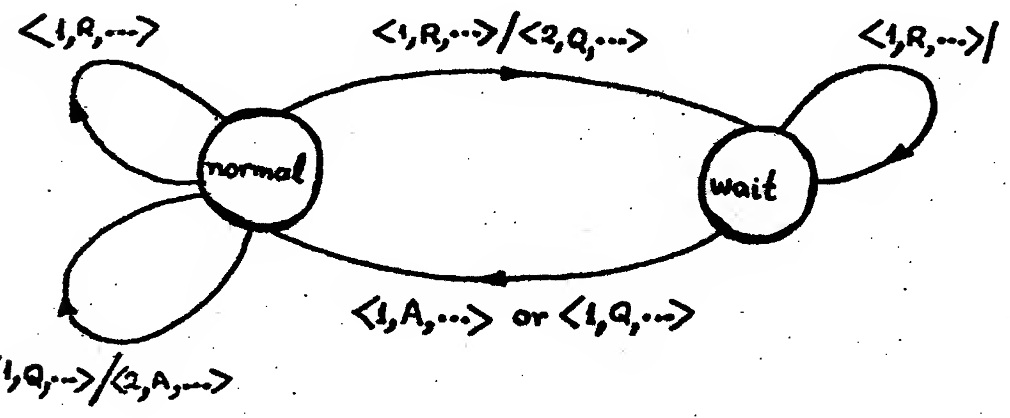


Figure 3.3 The mode at 1

Let *invlen*($\langle T,\alpha\rangle$)={number of actions in T}-{number of actions in α}

For the conditions (I) and (II) of Theorem 3 we have proven that (I) implies (II). We will prove that (II) implies (I) by induction on *invlen.*

Induction hypothesis: For *invlen*($\langle T,\alpha\rangle$)=j we have that, if (II) is true then (I) is true and moreover after $\langle T,\alpha\rangle$ has been executed LOCALSCHED$_i$($\langle T,\alpha\rangle$,b), i=1,2 realizes C and *sends* at most b messages.

For j=0 this is trivially true since $\langle T,\alpha\rangle$ is a history in C, there are no more requests left and $\langle T,\alpha\rangle \in M_C(0) \subseteq M_C(b)$. So we assume the hypothesis is true for j<j* and (II) is true for $\langle T,\alpha\rangle$ with *invlen* j* and some b (that depends on $\langle T,\alpha\rangle$). Since we have to prove (I), we have to exhibit a realization of C that *after* $\langle T,\alpha\rangle$ *sends* b or fewer messages. We consider the scheduler Q that realizes C by submitting all requests to one site, except when the input prefix $\langle T,\alpha\rangle$ has been executed. From that moment on Q uses LOCALSCHED$_i$($\langle T,\alpha\rangle$,b), i=1,2. We need only consider the operation of the scheduler after $\langle T,\alpha\rangle$. There are two cases:

Case A: h$\in$ C. First we will examine the case where no *send* instructions are executed and then the case where some are executed.

A.1: No *send* instructions are executed. Then the output has to be h and no request p waits for the execution of a request which is not an ancestor of p in h (Def. 12 is satisfied). This is because on every request p the test (Is new state in PR(C)?) is always true and involves only local computation. The reason for this is that by definition of C, as a concurrency control principle, h has no crossedges that are not forced by the transactions. Thus the part of the input each scheduler sees is automatically a prefix of h. Therefore it is unnecessary to wait for a message from the other site to verify that what the local scheduler sees is indeed a prefix of PR(C). Finally note that b$\geq$0.

A.2: Two or more *send* instructions are executed (the first two resulting in an exchange of a Q and an A message or two Q messages). Up to the first exchange the previous arguments, of A.1, hold. In order to execute *send* instructions a prefix β must exist that satisfies the conditions (1),(2) of Theorem 3 and has the new state t_i of a scheduler process as a projection.

Also t_i must be in $M_C(b\text{-}2)$, which can be decided since *invlen*(t_i)<j* (by the induction hypothesis and the "only if" part of the proof). Finally since $M_C(b\text{-}2)$ is not empty b≥2. After the exchange LOCALSCHED$_i$(s_{new},b-2) i=1,2 is used and we can invoke the induction hypothesis since *invlen*(s_{new})<j*. So h is outputed on-line with at most 2+(b-2) *send* instructions after <*T*,α>.

Case B: h∉ C. First we will prove that the output of the scheduler Q is a history in C (B.1). Finally that no more than b *send* instructions are executed (B.2).

B.1: Let the output (the granted requests) be a history h* not in C. Then it has (perhaps more than one) prefixes, called γ, such that γ∉ PR(C), γ has <*T*,α> as prefix and γ is minimal (all its prefixes are in PR(C)). Let us call q_1 and q_2 the final actions of γ, not in <*T*,α>, which are at sites 1 and 2 respectively. At least one of them must exist. Without loss of generality let site 1 grant q_1 before site 2 grants q_2 (if γ has a q_2). Since γ is minimal we have that either q_2 does not exist, or q_1 is an ancestor of q_2 in h* or q_1 and q_2 are concurrent in h* and then γ is an example of a β prefix of Theorem 3. If q_2 does not exist then, when the process at site 1 receives q_1 it cannot grant it, because it sees from the information available to it that the result would be incorrect. If q_1 is an ancestor of q_2 in h*, (that is there is a transaction crossedge making q_1 an ancestor of q_2 in h*) then site 2 knows q_1 has been executed (through a transaction defined message) and delays q_2. Finally if γ is an example of a β prefix of Theorem 3, then some exchange of two messages has to take place before q_2 and q_1 are granted. By (II) one of the projections of γ is in $M_C(b\text{-}2)$, b≥2, and thus, before both requests are ganted, one of the processes sends a Q message. If this exhange results in LOCALSCHED$_i$(s_{new},b-2) i=1,2 being initiated we can use induction to argue that γ cannot have been executed. If the exchange results in Delay*$_i$ i=1,2 being called, both processes output a correct predetermined completion of a common state s*. Thus we conclude that γ cannot have been executed and the output of the scheduler is always in C.

B.2: Since b≥0, if no *send* instructions are executed we have no problem. If *send* instructions are executed, let us look at the first round of communication (two Q messages or one Q and one A message). If as a result of this exchange LOCALSCHED$_i$(s_{new},b-2) i=1,2 is initiated with s_{new}∈

M_C(b-2), we know that $invlen(s_{new}) < j^*$ and $b \geq 2$ (See A.2). By induction no more than b-2 *send* instructions are used after this and again our goals are met. If as a result of this exchange $Delay^*_i$ i=1,2 is initiated at both sites (which is possible since the input $h \notin C$), then we know that $b \geq 2$. This is because (II) holds and a "bad" β exists. After both sites call $Delay^*_i$ they have a common state s^* and use no more *send* instructions, because the completion of s^* is predetermined and can be recognized locally. Thus no more than b *send* instructions are ever executed.

This completes the proof of Theorem 3.□

Corollary 3.1: If C, a concurrency control principle, has a computationally efficient realization, then it has a communication-optimal realization, which can be implemented in space polynomial in n (n=number of actions of $\mathcal{T}$).

Proof: It follows from Theorem 1 that, since C has a computationally efficient realization, recognizing if a prefix is in PR(C) can be done in polynomial time in n. Consider the following realization Q:

Q: (1) Each site computes b^* from $\mathcal{T}$, where $b^* = b^*(\mathcal{T}) = \min\{b / \langle \mathcal{T}, \emptyset \rangle \in M_C(b)\}$
(2) Site i uses $\text{LOCALSCHED}_i(\langle \mathcal{T}, \emptyset \rangle, b^*)$ (i=1,2)

By the constructive proof of Theorem 3 Q is a realization of C using the minimum (b^*) number of messages. From this proof we have that four computational tasks are performed by LOCALSCHED. These are:

(a) Given t, does $t \in$ PR(C)?
This can be performed in polynomial time (and therefore space).

(b) Given $t \in$ PR(C), $i \neq j$, and $r \leq t$, is there a β such that:
$\{t = (\beta/r)_i\} \wedge \{\beta \notin \text{PR(C)}\} \wedge \{(\beta/r)_j \in \text{PR(C)}\}$?
This can be performed in nondeterministic polynomial time (and therefore space).

(c) Given $t \in$ PR(C), $b \geq 0$, does $t \in M_C(b-2)$?
Using Theorem 3, the polynomial characterization of PR(C) and the theory of alternation [5], we have that both this task and step (1) of Q can be implemented in polynomial space.

(d) Finally if Q discovers that the input is incorrect and $Delay^*_i$ is called at both sites then a correct completion of s^* can be efficiently computed. This can be done based on a predetermined ordering of the actions and the test of membership in PR(C).

This completes the proof of the existence of a communication-optimal scheduler realizing C in polynomial space and exponential time.□

We will end this section with some comments on Theorem 3.

(1) Message lengths: Let us examine the length in bits of the messages sent. If $|T|=n$ there are at most n! states and in order to uniquely code a state we need O(nlogn) bits. Also we never send more than 2n messages. In the proof of Theorem 3 we have used an inefficient format for messages $\langle ...,s_i \rangle$. Although for clarity of presentation we used s_i (O(nlogn) bits) in our messages, we could have as well used $s_i \backslash r_i$ (i.e., each site will hear of every action at most once). Thus in total O(nlogn) bits will be used in the worst case.

(2) More than two sites: The two site case, while being the simplest distributed configuration is sufficient for the results of Chapter 4. If more sites are used and the mode of communication is a broadcast mode, Theorem 3 can be easily generalized. On the other hand a network of sites makes optimal communication a more difficult problem, since it implicitly adds the problem of appropriately routing the messages.

(3) Persistency: We have examined schedulers that realize C and consist of two processes, one at each site. Each of these processes knows of some *pending requests* and a prefix of a history in C that has been executed (its *state*).

We call such realizations of C *persistent* if whenever a process i discovers that the execution of a pending request p_i would make its state s_i incorrect, it delays p_i indefinitely and proceeds as if only the requests in s_i had been submitted.

If PR(C)$\in$ P there are persistent polynomial time schedulers realizing C, as is obvious from the proof of Theorem 1 and [25]. On the other hand the scheduler of Corollary 3.1 is not persistent. For some incorrect inputs Delay* is used. This is because persistency requires that messages are sent even after the input is discovered to be incorrect. To illustrate this suppose our scheduler starts with $\langle T,\alpha \rangle \in M_C(b)$ and receives a "bad" input $\langle T,\beta \rangle$ with projections $\langle T,\alpha_1 \rangle \in M_C(b\text{-}2)$ and $\langle T,\alpha_2 \rangle \notin M_C(b\text{-}2)$.

It is possible for $\langle T,\alpha_2\rangle$ to have been executed when the scheduler, at the expense of two messages, discovers the input to be incorrect. If we want our scheduler to be persistent, starting from $\langle T,\alpha_2\rangle$ it has to use more than b-2 *send* instructions.

This difference between on-line computationally efficient and on-line communication efficient algorithms, which accept the same strings, arises because of the nature of resources we are trying to optimize. In one case we wish to achieve performance C at asymptotic computation cost $O(n^k)$, in the other at fixed (say n/15 or 200) communication cost.

From the proof of Theorem 3 it is easy to see that:
"$\langle T,\alpha\rangle \in M_C(b)$ iff there is a *persistent* realization of C, which *if the input is in* C *sends* at most b messages after $\langle T,\alpha\rangle$".

We have related communication complexity of schedulers achieving parallelism C, with the computational problems $\langle T,\alpha\rangle \in M_C(b)$? (which are in ***PSPACE***).

If the input history is in C and $\langle T,\emptyset\rangle \in M_C(b)$ a user's delay D is bounded by:
b(communication delay/message) $\geq$ D ≥ 0.
If the input history is not in C there is a user who has to wait for other users.

The approach of Theorem 3 and the formulation of the scheduling problem are pretty much independent of concurrency control and serializability. The application to databases provides practical motivation and analytical tools (i.e., mixed ordered multigraphs). In fact the entire methodology can be extended to distributed ***on-line*** computation of combinatorial functions of two integers, which in a distributed environment are stored at two different sites [38].

3.2 Games related to Distributed On-line Computation

In this section we will define decision problems for the sets of prefixes, which were recursively characterized in the previous section.

Distributed scheduling is related, using $M_C(b)$, to a game on prefixes, **PREFIX**, whose rules are displayed in Fig. 3.4. In this game Player I corresponds to a malicious adversary who wishes to force communication. His move is a "bad" continuation β of the current position α. Player II corresponds to the two cooperating scheduler processes. Each one of his choices i* indicates, which of the two processes has the responsibility of guarding against the "bad" continuation β (by questioning the other process before proceeding). Player I wants to prolong the game as much as possible, whereas Player II tries to bring it to an end as soon as possible (other than that there is no winner or looser).

From Theorem 3 we can deduce the following property of communication-optimal realizations of C:

Corollary 3.2: The minimum number of messages sent by a communication-optimal realization of C equals the length of PREFIX($\langle T,\emptyset\rangle$) if both players play optimally (we call this the minimax length of PREFIX).

Proof: Follows from Theorem 3 and the theory of alternation [5]. Note that although in general we define PREFIX from an arbitrary initial position $\langle T,\alpha_0\rangle$, we are in fact interested in $\alpha_0=\emptyset$. T represents the static (a-priori) information on transaction schemata, that is used to optimize communication. Thus $\{\langle T,\alpha\rangle \notin M_C(b)?\}$ is equivalent to {Is the minimax length of PREFIX($\langle T,\alpha\rangle$) greater than b ?}.□

In the following section we will analyze the game PREFIX for C=SR. If we choose serializability (SR) as our concurrency control principle the board position becomes the conflict graph G(T) with some of its edges directed. The moves of Player I become choices of directions to undirected edges of G(T). Much insight into PREFIX in this case is gained by studying a game played on mixed graphs called CONFLICT and displayed in Fig. 3.5. This game is our departure point in the PSPACE-Completeness proof, given in the next section.

PREFIX($\langle T,\alpha_0\rangle$)

Initial position: For fixed C, a prefix $\langle T,\alpha_0\rangle$

Position before player I's move: A prefix $\langle T,\alpha\rangle$

Player I's move: Select a prefix $\langle T,\beta\rangle$ *such that*

> *(1)* β is a continuation of α, with projections $\alpha_i=(\beta/\alpha)_i$ $i=1,2$
> *(2)* α_1,α_2 are prefixes of C
> *(3)* β is not a prefix of C

Player II's move: Select $i^* \in \{1,2\}$ *and set* $\alpha:=\alpha_{i^*}$

Players I and II take turns moving. Player II always moves when I does.
Player I's goal is to prolong the game as much as possible.
Player II's goal is to end the game as soon as possible.

Figure 3.4
The game PREFIX

CONFLICT(G_0)

Initial position: A mixed graph $G_0=(V_0,E_0,A_0)$
(E_0 partitioned into "red" and "green")

Position before player I's move: A mixed graph $G=(V,E,A)$

Player I's move: Select an assignment of directions (A_X) to an $X \subseteq E$ *such that*
(1) R(H) is the "red"("green") subset of X
(2) $A_R \cup A$, $A_H \cup A$ have no directed cycles
(3) $A_X \cup A$ has a directed cycle

Player II's move: Select $Y \in \{R,H\}$ *and set* $E:=E \setminus Y$ and $A:=A \cup A_Y$

Players I and II take turns moving. Player II always moves when I does.
Player I's goal is to prolong the game as much as possible.
Player II's goal is to end the game as soon as possible.

Figure 3.5
The game **CONFLICT**

CONFLICT$^+(G_0)$

Initial position: An ordered mixed multigraph $G_0=(V_0,E_0,A_0,\{\geq_i\})$
(E_0 partitioned into "red" and "green")

Position before player I's move: An ordered mixed multigraph $G=(V,E,A,\{\geq_i\})$

Player I's move: Select a closed assignment (A_X) to an $X\subseteq E$ *such that*

(1) A_X has projections $A_X{}^r, A_X{}^g$

(2) $A_X{}^r\cup A$, $A_X{}^g\cup A$ have no directed cycles

(3) $A_X\cup A$ has a directed cycle

Player II's move: Select $y\in\{r,g\}$ *and set* $E:=E\backslash(\text{edges in } A_X{}^y)$ and $A:=A\cup A_X{}^y$

Players I and II take turns moving. Player II always moves when I does.
Player I's goal is to prolong the game as much as possible.
Player II's goal is to end the game as soon as possible.

Figure 3.6
The game CONFLICT$^+$

The game CONFLICT abstracts, in the legal moves of Player I, only the rules of PREFIX derived from an unordered conflict graph (β has to create a cycle in the conflict graph, while $(\beta/\alpha)_i$ $i=1,2$ should not). In fact the assignments of directions to edges of $G(\mathcal{T})$ in PREFIX should also correspond to prefixes β and $(\beta/\alpha)_i$ $i=1,2$ (see Lemma 1, Section 2.2). CONFLICT can obviously be played on multigraphs with no modifications of its rules.

We will now generalize the game CONFLICT to CONFLICT$^+$ (see Fig. 3.6), where in addition to the rules of CONFLICT a precedence rule is observed.

The input to the new game CONFLICT$^+$(G) is an ordered mixed multigraph $G=(V,E,A,\{\geq_i\})$. (V) is the vertex set, (E) is the multiset of undirected edges partitioned into "red" and "green", (A) is the multiset of directed edges and $\{\geq_i\}$ are partial orders (e.g. all undirected edges incident at node i form a partial order $\geq_i$). All conflict graphs (see Def. 9, Section 2.1) are such constructs. If $A \neq \emptyset$ some conflicts have been resolved and the $\geq_i$'s correspond to transaction partial orders.

<u>Definition 15</u>: Given an ordered mixed multigraph $G=(V,E,A,\{\geq_i\})$, and an assignment (A_X) of directions to a multiset of edges $X \subseteq E$, we call this assignment **closed (with respect to G)** when:

If $ij \in X$ and is directed from i to j and $ik \geq_i ij$ then $ik \in X$.□

Given a conflict graph $G(\mathcal{T})$ and an assignment of directions to some of its edges (A_X), that has no directed cycles, then A_X is realizable by a prefix in SR iff it is closed. This follows easily from Theorem 2 and Lemma 1 (see Section 2.2).

Let the undirected edges of G be partitioned into "red" and "green", and let A_X be a closed assignment of directions to $X \subseteq E$. It is easy to see that the following closed assignments are uniquely determined. They are called the *projections* of A_X.

A_X^i (where i="red" or "green"):

(1) $A_X^i \subseteq A_X$

(2) A_X^i is closed

(3) all i edges of X are given directions in A_X^i

If $\{\geq_i\}$ become the empty partial orders for every node, CONFLICT$^+$ becomes CONFLICT (i.e., $X=R\cup H$, $A_X{}^r = A_R$, $A_X{}^g = A_H$). The real interest of CONFLICT$^+$ is its relation to PREFIX. A prefix $\langle T,\alpha\rangle$ in PR(C) determines a unique mixed ordered multigraph $G^\alpha(T)$ (see Def. 10, Section 2.1). In the next section (Lemma 2, Section 4.1) we will show that for C=SR, PREFIX($\langle T,\alpha\rangle$) and CONFLICT$^+$($G^\alpha(T)$) are equivalent. An example of CONFLICT, where an optimal game leads to four moves is presented in Fig. 3.7.

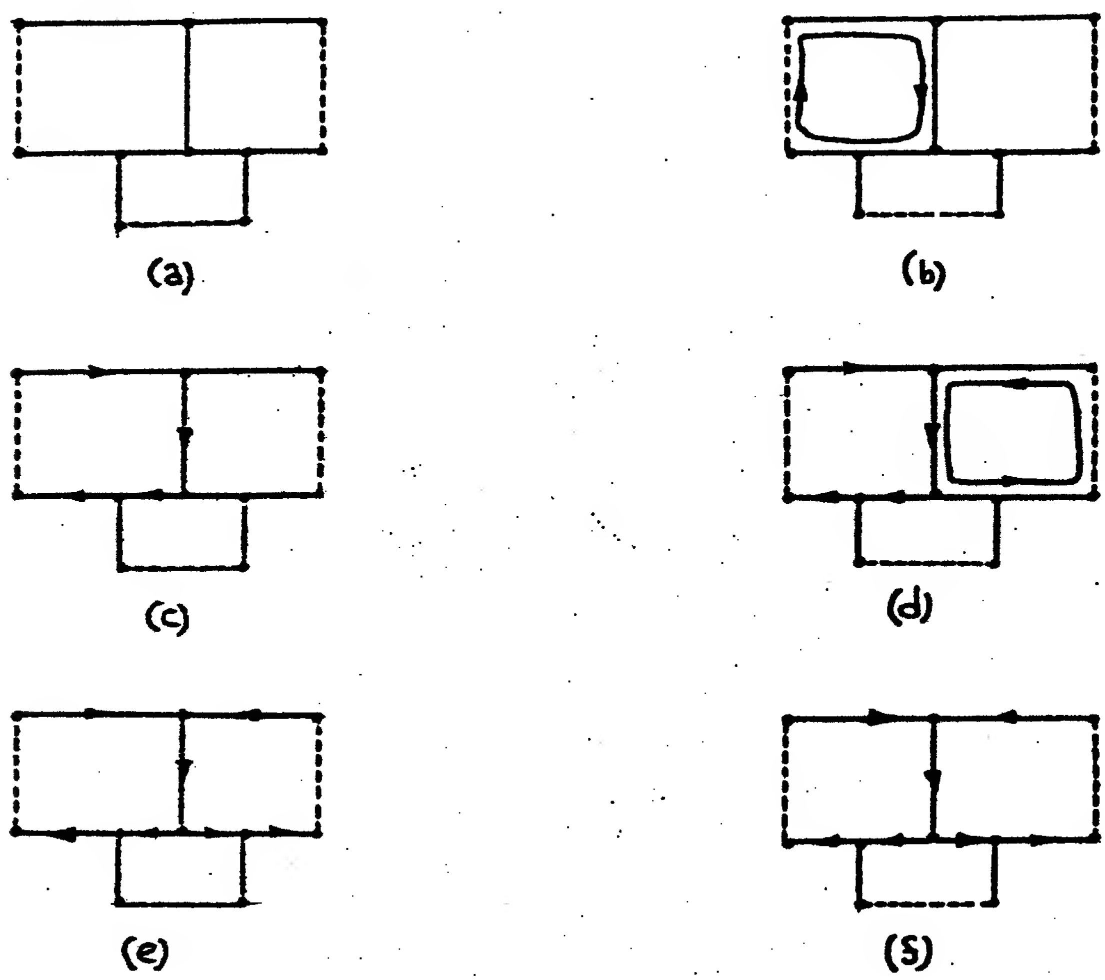


Figure 3.7

(a) $G(T)$ initial position (—— "red", - - - "green")

(b) I's first move

(c) II chooses "red"

(d) I's second move

(e) II chooses "red"

(f) no legal moves for I

We will close this section with a brief discussion of an important special case of the question $\{\langle T,\alpha\rangle \in M_c(b)?\}$ namely b=0. This problem is obviously in *NP*, because all we have to do is guess a prefix satisfying conditions (1),(2) of Theorem 3 and check these conditions in polynomial time.

In the next section (Corollary 4.2, Section 4.1) we will prove that $\{\langle T,\alpha\rangle \in M_c(0)?\}$ is *NP-Complete*. This leaves us with the problem $\{\langle T,\emptyset\rangle \in M_c(0)?\}$. We say that the conflict graph G(T) of a transaction system T contains a *mixed cycle*, if it contains a cycle with edges e_1 and e_2, where e_1 corresponds to a conflict at site 1 ("red") and e_2 to a conflict at site 2 ("green").

Corollary 3.3: For C=SR, if G(T) contains no mixed cycle then $\langle T,\emptyset\rangle \in M_c(0)$. This is also a necessary condition, whenever the transactions in T have no crossedges.

Proof: The sufficiency is obvious from the characterization of SR and conditions (1),(2) of Theorem 3. The necessity for transactions with special structure is easy for two transaction systems. For more transactions we can use a straightforward induction on the number of transactions (nodes of G(T)).□

For general transaction systems T and C=SR, the complexity of determining if $\{\langle T,\emptyset\rangle \in M_c(0)?\}$ is an interesting open question. For example all systems in Fig. 3.8 are in $M_c(0)$, yet their conflict graphs contain mixed cycles.

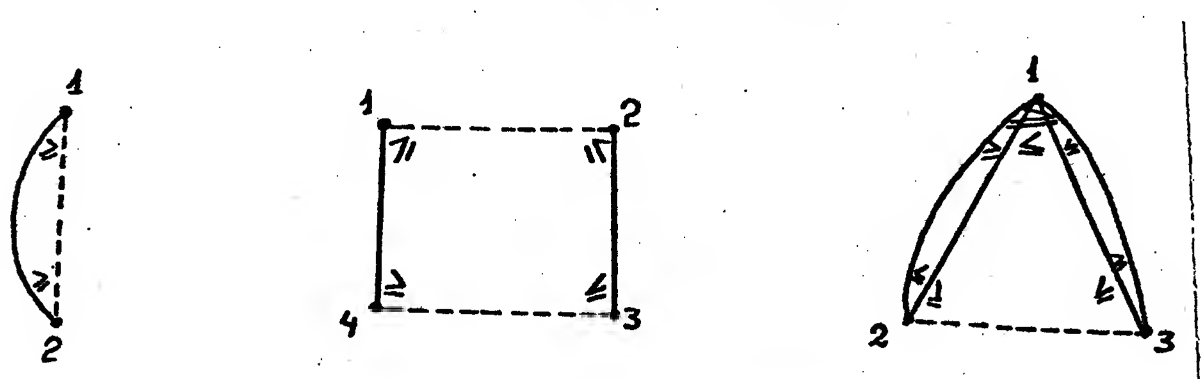


Figure 3.8 G(T)'s for $\langle T,\emptyset\rangle \in M_c(0)$.

4. The Complexity of PREFIX

This chapter contains our main result, which is an analysis of the game PREFIX for C=SR.

4.1 PREFIX is PSPACE-Complete

We will now prove the following theorem:

Theorem 4: Let C=SR. For input T and $b \geq 0$, determining whether the minimax length of PREFIX($\langle T, \emptyset \rangle$) is greater than b is ***PSPACE-Complete***.

All the games we will examine in this section are in ***PSPACE***. This follows easily from the analysis in Chapter 3. Therefore we will present only the reduction of a well known ***PSPACE-Complete*** problem to PREFIX. This is the problem QBF (i.e., what is the truth value of a quantified boolean formula)[11,33,34].

QBF:
Input: A quantified boolean formula I_n of the form:
$\exists x_1 \forall x_2 \exists x_3 \ldots \exists x_{n-1} \forall x_n \; F(x_1,x_2,\ldots,x_n)$
where F is a boolean formula without quantifiers in 3CNF (3-conjunctive normal form) of the variables $x_1,\ldots,x_n$ (n=even).
Question: Is I_n true?

QBF can be viewed as a game between two players, the $\exists$-player and the $\forall$-player. These players take turns assigning values to the variables in the order these variables are quantified in I_n (i.e., from left to right). First the $\exists$-player assigns a value to x_1, then the $\forall$-player assigns a value to x_2 etc. The $\exists$-player wins if the values assigned to the x_i's $i=1,\ldots,n$ make $F(x_1,x_2,\ldots,x_n)$ true, otherwise he looses. The $\exists$-player has a winning strategy iff I_n is true. This ***PSPACE-Complete*** problem is used in most reductions to games, [5,11,33,34,8,29].

Another game on boolean formulas used in our proof is AE-QBF. This is similar to QBF only the ∀-player makes all his moves before the ∃-player.

<u>AE-QBF</u>:

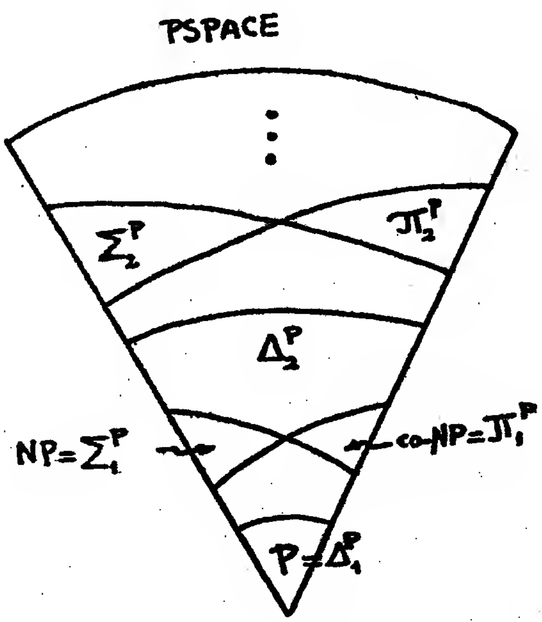
<u>Input</u>: A quantified boolean formula I_n of the form:

$\forall x_2 \forall x_4 \ldots \forall x_n \exists x_1 \exists x_3 \ldots \exists x_{n-1}\ F(x_1,x_2,\ldots,x_n)$

where F is a boolean formula without quantifiers in 3CNF (3-conjunctive normal form) of the variables $x_1,\ldots,x_n$ (n=even).

<u>Question</u>: Is I_n true?

AE-QBF is Π_2^P*-Complete*, where Π_2^P is a class of the polynomial time hierarchy [33,11] corresponding to one ∀∃ alternation (see Fig. 4.1).



$\Sigma_0^P=\Pi_0^P=\Delta_0^P=P$

for all $k\geq 0$

$\Delta_{k+1}^P=P^{\Sigma k}$

$\Sigma_{k+1}^P=NP^{\Sigma k}$

$\Pi_{k+1}^P=co\text{-}\Sigma_{k+1}^P$

Figure 4.1

The polynomial time hierarchy

P^Y={L: there is a language L'∈ Y s.t. L is P- time Turing reducible to L'}

NP^Y={L: there is a language L'∈ Y s.t. L is NP- time Turing reducible to L'}

Our reduction of QBF to PREFIX will proceed in four parts, which we outline below from (I) to (IV).

(I) We show that CONFLICT, as defined in Fig. 3.5, is Π_2^P*-hard*. We accomplish this by reducing AE-QBF to CONFLICT in Lemma 1. The input graphs to CONFLICT are mixed (i.e. they may contain directed edges).

(II) We generalized CONFLICT to the game CONFLICT$^+$, that has in addition to the rules of CONFLICT a partial order on edges incident at a node. The definition is such that all possible conflict graphs $G(T)$ can be inputs to CONFLICT$^+$. In Lemma 2 we prove that the game PREFIX (for C=SR) is a special case of CONFLICT$^+$.

(III) We prove that CONFLICT$^+$ is ***PSPACE-Complete***, even when the input is a graph without directed edges. We accomplish this in Lemma 3 using many of the constructs of Lemma 1.

(IV) Finally we prove that PREFIX($\langle T,\emptyset\rangle$) is ***PSPACE-Complete*** by showing that the graphs in Lemma 3 are in fact conflict graphs for some transaction system.

In Lemma 1 we will examine the game of CONFLICT (see Fig. 3.5). Its input is a mixed graph G=(V,E,A), where E is partitioned into "red" and "green". Player I picks an assignment of directions for a "red" subset of $E(A_R)$ and for a "green" subset of $E(A_H)$. The choices he makes must be legal (i.e. $A \cup A_R, A \cup A_H$ have no directed cycles, $A \cup A_R \cup A_H$ has a directed cycle). Player II chooses "red"("green") making the new directed board position $A \cup A_R (A \cup A_H)$ from A. Player I wants to make the game last and Player II wants to terminate it.

The direction of an undirected edge e can become *fixed* during the game in two ways. First if Player I chooses e as part of $A_R(A_H)$ and Player II chooses "red"("green"). After this e becomes part of A, the directed section of the board position. On the other hand, even if e has not become part of the directed (A) before Player I makes his new move, it is possible for A to contain a directed path between the endpoints of e. Now e's direction is fixed, because it can only be used in one direction, if Player I's moves are to be legal. It is easy to see that if a move by Player I is legal $A_R(A_H)$ must contain edges, whose directions have not been fixed. Because of this observation the following fact is easily seen to be true.

(0) *If G has z "green" edges CONFLICT(G) lasts at most 2z moves.* If Player I makes a move with two "green" edges, whose directions have not been fixed, a move of "green" by Player II would consume two "green" edges. Moreover if Player I makes a move with exactly one "green" edge (e), whose direction has not been fixed, then no matter what the response of Player II is e's direction becomes fixed (i.e., either e becomes part of the new A or a path is included in the new A connecting the endpoints of e).

We will use the notation MN for an undirected edge and (MN) for a directed edge from M to N. Similarly $M_1M_2...M_k$ will be an undirected and $(M_1M_2...M_k)$ a directed path from M_1 to M_k.

Lemma 1: Given a mixed graph G and a nonnegative integer b, determining whether the minimax length of CONFLICT(G) is greater than b is Π_2^P*-hard.*

Proof: For an arbitrary instance I_n of AE-QBF we construct the mixed graph $G(I_n)$ using the rules (a) to (d) below. We will prove that I_n is true iff the game CONFLICT can last more than n moves on $G(I_n)$.

(a) For every existentially quantified variable x_i, $i=1,3,\dots,n-1$ in I_n a copy of the graph in Fig. 4.2(c) is included as a subgraph of $G(I_n)$. This subgraph contains 6 directed edges and 2 "red" undirected edges, namely T_iD_i (labelled with 1) and F_iE_i (labelled with 0). Actually this is the graph of Fig. 4.2(b) without nodes A_i,B_i,M_i,N_i. These are the $\exists$-subgraphs.

(b) For every universally quantified variable x_i, $i=2,4,\dots,n$ in I_n a copy of the graph in Fig. 4.2(a) is included as a subgraph of $G(I_n)$. This subgraph contains 6 directed edges, 1 "red" undirected edge T_iD_i (labelled with 1) and 1 "green" undirected edge F_iE_i (labelled with 0). These are the $\forall$-subgraphs.

(c) For every clause of the 3CNF formula of I_n (i.e. $F(x_1,x_2,\dots,x_n)$) a copy of the graph in Fig. 4.3 is included as a subgraph of $G(I_n)$. This subgraph contains 35 directed edges and 21 "red" undirected labelled edges. For the kth clause $(u \vee v \vee w)$, (starting from left to right in $F(x_1,x_2,\dots,x_n)$), which has literals u,v,w, we have seven possible paths from C_k to C_{k+1}. Each one of these paths corresponds to an assignment of values to the literals u,v,w, of the clause, which makes the clause true (i.e. only assignment 000 is excluded). The assignment can be read from the labels of "red" edges on the path. Every one of the three columns, of seven labels each, corresponds to the possible values of one literal. Also for one literal (say u) four directed edges go to F_u and three to T_u, depending on the label of the "red" edge from which the directed edge starts. We call these directed edges (to F_u or T_u) *backedges.* We use the following rule:

$u=x_i \Rightarrow F_u=F_i$ and $T_u=T_i$

$u=\neg x_i \Rightarrow F_u=T_i$ and $T_u=F_i$ for x_i a variable of I_n

The backedges are connected so that if the labels correspond to values of variables and literals a backedge connects two undirected edges iff their labels are inconsistent (e.g. $x_1=1$, $u=\neg x_1$, a backedge connects T_1D_1 and "red" edges with labels 1 in the column of u). These are the clause-subgraphs.

(d) The graph $G(I_n)$ is constructed by identifying nodes with the same name. That is S_p's of ∃-subgraphs with S_q's of ∀-subgraphs if $p=q$. Also F_p's or T_p's of ∃- and ∀-subgraphs are identified with F_q's or T_q's of clause-subgraphs if $p=q$. We also identify $C_1=S_{m+1}$. If there are m clauses in I_n we add the "green" edge S_1C_{m+1}.

An example is provided in Fig. 4.4 for the AE-QBF: $I_n=\forall x_2 \forall x_4 \exists x_1 \exists x_3\ (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_4 \vee \neg x_3)$, if we delete the nodes A_i, B_i, M_i, N_i, $i=1,3$ and A_{13}, A_{12}, A_{34}. We will first examine some simple properties of $G(I_n)$.

(1) *Let $G(I_n)$ contain z "green" edges. Then CONFLICT($G(I_n)$) can last 2z-2 moves and at most 2z moves.* Here $z=n/2+1$. The game can last 2z moves, because of observation (0) (right before Lemma 1). It can last always 2z-2 moves, because Player I can play z-1 times on the $z-1=n/2$ mixed cycles $(F_iE_iT_iD_iF_i)$, $i=2,4,...,n$. His moves are legal no matter what the response of Player II is.

(2) Let $(S_1...C_{m+1})$ be any directed path from S_1 to C_{m+1}, not using the "green" edge S_1C_{m+1} and respecting the directed edges in $G(I_n)$. We note that each pair F_iE_i, T_iD_i, $i=1,2,...,n$, forms a cutset separating S_1 and C_{m+1}. *Thus $(S_1...C_{m+1})$ contains F_iE_i or T_iD_i for all $i=1,2,...,n$.*

(3) All paths $(S_1...C_{m+1})$ have to contain node C_1. If they contain a *backedge* it is easy to see that they have to pass through C_1 at least two times. *Therefore simple paths (containing a node only once) $(S_1...C_{m+1})$ do not contain backedges.*

Let us proceed with the proof of equivalence:

"only if" If I_n is true then Player I first makes n/2 moves on the ∀-subgraphs using the mixed cycles $(F_iE_iT_iD_iF_i)$, $i=2,4,...,n$. The n/2 moves of Player II fix directions for all the undirected edges F_iE_i, T_iD_i $i=2,4,...,n$. His choice of "red" turns T_iD_i into (T_iD_i) and fixes the direction of F_iE_i to (E_iF_i), (because of the directed path $(E_iT_iD_iF_i)$, which now becomes part of A). This corresponds to assigning x_i the value 1. Similarly his choice of "green" turns F_iE_i into (F_iE_i) and fixes the direction of T_iD_i to (D_iT_i). This corresponds to assigning x_i the value 0.

At this point in CONFLICT 2z-2 moves have been made and we can say that the choices of Player II have assigned values x^*_i to the variables x_i, $i=2,4,...,n$. Since I_n is true there exist values x^*_i of the variables x_i, $i=1,3,...,n-1$, which make

$F(x^*_1, x^*_2, ..., x^*_n)$ true. This assignment of values $\{x^*\}$ to variables $\{x\}$ implies an assignment of values to the literals of every clause $\{u(x^*)\}$ (e.g., $u=\neg x$, $x^*=1$ implies $u^*=0$).

Let us describe the n/2+1st move of Plater I. Consider the simple path $(S_1...C_{m+1})^*$, which consists of the following subpaths in the various subgraphs of $G(I_n)$.

$(S_kT_kD_kS_{k+1})$ if $x^*_k=1$, $k=1,2,3,...,n$. In $\forall$-subgraphs the direction of T_kD_k has been fixed to (T_kD_k). In $\exists$-subgraphs (T_kD_k) is used.

$(S_kF_kE_kS_{k+1})$ if $x^*_k=1$, $k=1,2,3,...,n$. In $\forall$-subgraphs the direction of F_kE_k has been fixed to (F_kE_k). In $\exists$-subgraphs (F_kE_k) is used.

In the kth clause-subgraph the path from C_k to C_{k+1}, whose labels are the values assigned to the literals of the clause by $\{x^*\}$. Such a path exists since no clause is assigned the values 000 by $\{x^*\}$.

We note that, because of the way $(S_1...C_{m+1})^*$ traverses $\forall$-$\exists$- and clause-subgraphs, the directed edges of $G(I_n)$ and $(S_1...C_{m+1})^*$ form no directed cycle. Note that no backedge has both its endpoints on $(S_1...C_{m+1})^*$, because the labels in the various subgraphs along $(S_1...C_{m+1})^*$ are consistent.

Using the rules of Fig. 3.5 Player I picks:

A "green" set $H=\{S_1C_{m+1}\}$ and directs it (A_H) from C_{m+1} to S_1.

A "red" set $R=\{$"red" edges in $(S_1...C_{m+1})^*\}$ and directs them (A_R) along the path $(S_1...C_{m+1})^*$.

This is a legal move since: $A_R \cup A$, $A_H \cup A$ are acyclic, $A_R \cup A_H \cup A$ is not. Therefore if I_n is true CONFLICT can last n+2 moves.

"if" If I_n is false we will prove that CONFLICT($G(I_n)$) cannot last n+2 moves. We will assume CONFLICT($G(I_n)$) can last n+2 moves and reach a contradiction.

The move of Player I, which has "green" edge $S_1C_{m+1} \in H$ must be his n/2+1st move. This is because, if the direction of some "green" edge has not been fixed yet, any simple path $(S_1...C_{m+1})$ that Player I chooses would make it possible to fix the directions for two "green" edges. This follows from property (2) of such paths, proven above, and the structure of the $\forall$-subgraphs. Thus Player I must make n/2 moves involving the "green" edges in the $\forall$-subgraphs first. Moreover any choice of Player II will fix their direction, (by observation (0)). We will prove that there is a

sequence of choices by Player II that will not let Player I move another time.

Since I_n is false then $\neg I_n$ is true or,

$\exists x_2 \exists x_4 ... \exists x_n \forall x_1 \forall x_3 ... \forall x_{n-1} \; \neg F(x_1, x_2, ..., x_n)$

Let the values of the x_i's, $i=2,4,...,n$ making this formula true be x^*_i. For the first n/2 moves of Player I, each one necessarily involving a single F_iE_i, whose direction has not been fixed, the response of Player II should be:

If $x^*_i=0$ then "green". This fixes the directions of T_iD_i and F_iE_i into (D_iT_i) and (F_iE_i) respectively.

If $x^*_i=1$ then "red". This fixes the direction F_iE_i into (E_iF_i).

The n/2+1 st move of Player I is now constrained in several ways in order to be legal. First for the "green" set we know $S_1C_{m+1} \in H$, because it is the only "green" edge, whose direction has not been fixed. Second for the "red" set we know that {undirected "red" edges of a path $(S_1...C_{m+1})$} $\subseteq$ R. Finally $(S_1...C_{m+1})$ and the directed part of $G(I_n)$ must not contain a cycle. This path $(S_1...C_{m+1})$ must be simple (no backedges by property (3)) and thus pass through all the subgraphs:

In a clause-subgraph it has to use one of the seven paths.

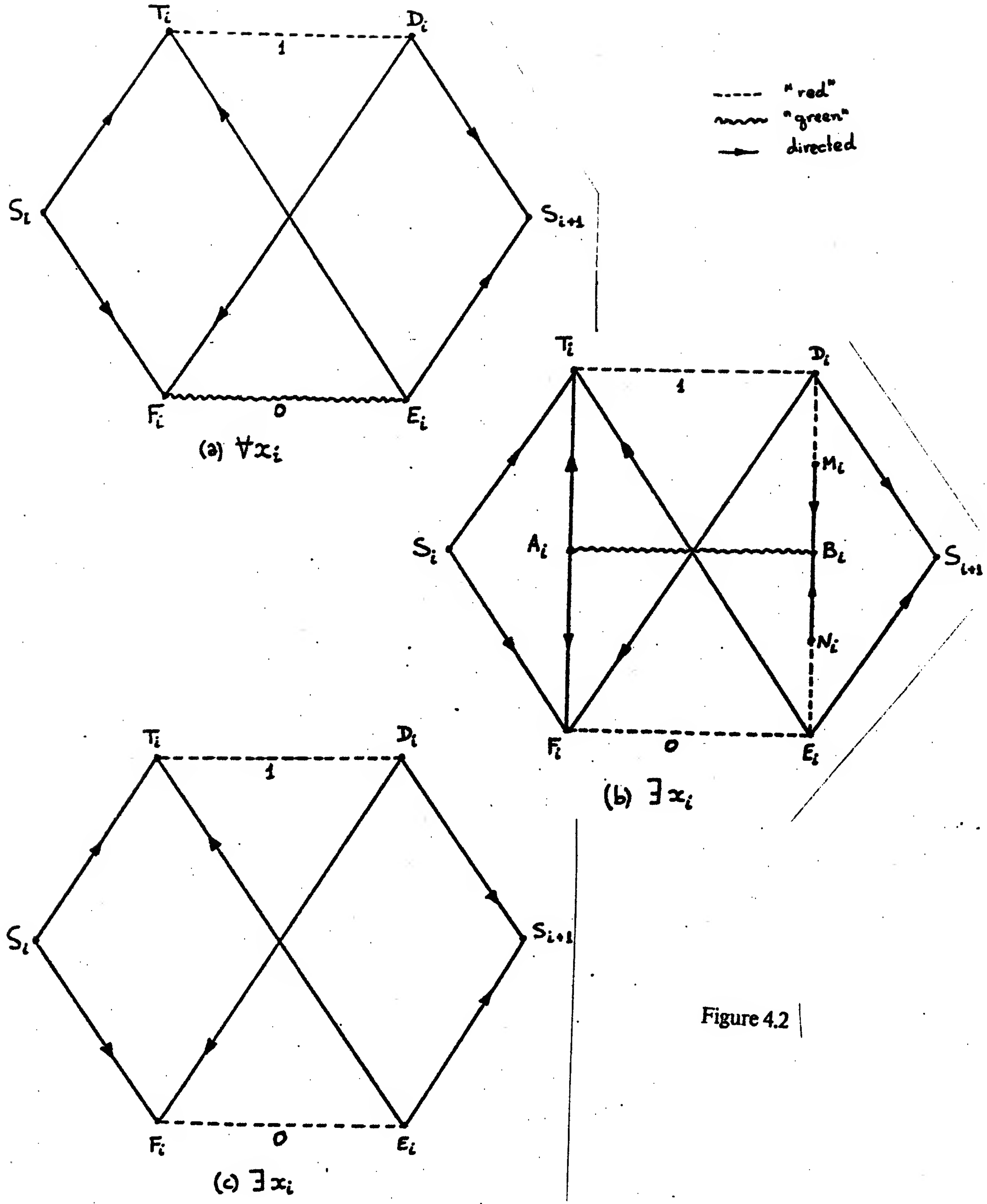
In a $\forall$-subgraph its behavior is constrained by the way the directions of edges T_iD_i, F_iE_i (of which it contains exactly one) have been fixed.

In a $\exists$-subgraph it is constrained to contain exactly one of T_iD_i, F_iE_i. Else $(S_1...C_{m+1})$ and the directed part of $G(I_n)$ would contain a cycle. We extend the assignment $\{x^*\}$ in the following way for $i=1,3,...,n-1$:

If $(S_1...C_{m+1})$ contains T_iD_i then $x^*_i=1$ else $x^*_i=0$.

Thus every candidate path $(S_1...C_{m+1})$ actually corresponds to an assignment $\{x^*\}$ of values to the variables and $\{u^*\}$ to the literals of F. This assignment can be read from the labels of edges along the path. In fact $\{x^*\}$ and $\{u^*\}$ are inconsistent.

By the way x^*_i, $i=2,4,...,n$ were chosen every candidate assignment makes $F(x^*_1,...,x^*_n)$ false. Thus a consistent assignment $\{u(x^*)\}$ to the literals must make the literals in some clause (say the kth clause) 000. Our candidate path $(S_1...C_{m+1})$ uses a subpath $(C_k...C_{k+1})$ in that clause, which has a label 1 for one of its literals. Because of the initial connection of backedges, the backedge of that literal ends at a node that belongs to the path $(S_1...C_{m+1})$ in a $\forall$- or $\exists$-subgraph. Therefore $A_R \cup A$ cannot be chosen to be acyclic and no candidate n/2+1st move of Player I can be legal. This is the desired contradiction. $\square$



T_i
D_i
1
S_i
S_{i+1}
F_i
0
E_i
(a) ∀x_i
"red"
"green"
directed
T_i
D_i
1
M_i
S_i
A_i
B_i
S_{i+1}
N_i
F_i
0
E_i
(b) ∃x_i
T_i
D_i
1
S_i
S_{i+1}
F_i
0
E_i
(c) ∃x_i

Figure 4.2

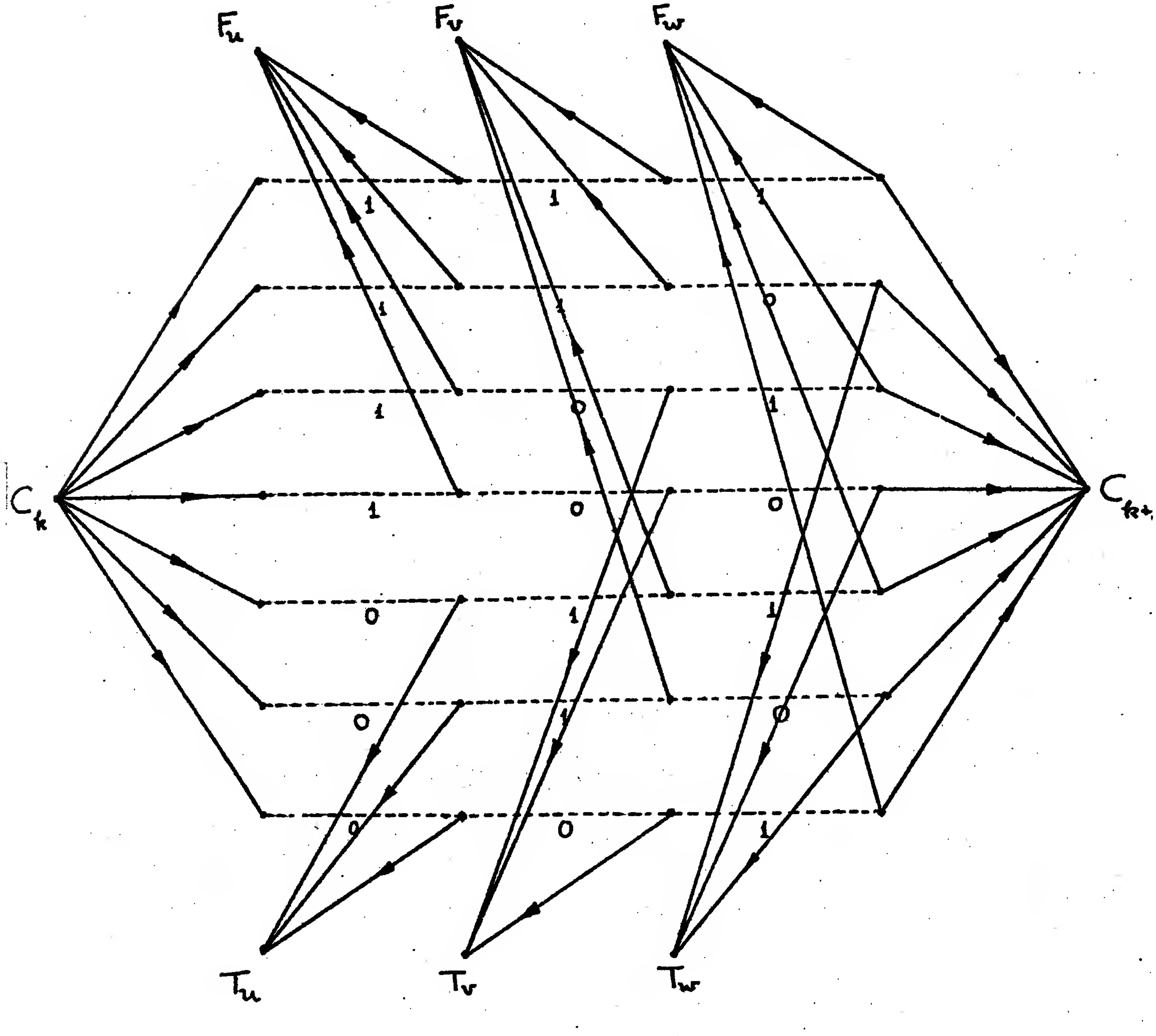
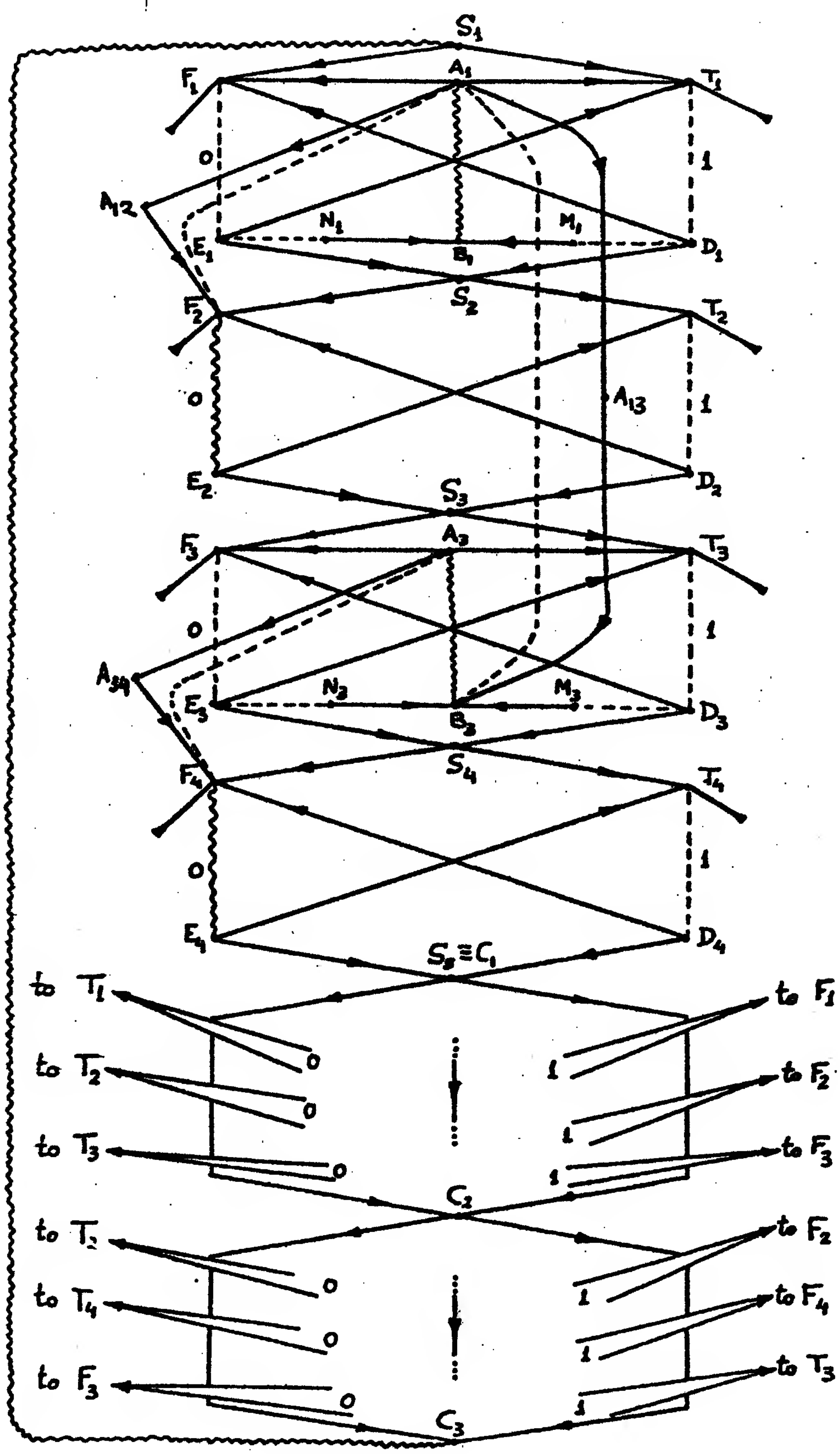


Figure 4.3 The kth clause subgraph

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \; (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_4 \vee \bar{x}_3)$$

Figure 4.4 An example



We will now examine the game CONFLICT$^+$(G), which has as input an ordered mixed multigraph $G=(V,E,A,\{\geq_i\})$. The edge multiset E is partitioned into "red" and "green". The undirected edges incident at node i belong to the partial order $\geq_i$. The game is like CONFLICT((V,E,A)) the only difference is that assignments A_X (corresponding to $A_R \cup A_H$), $A_X{}^r$ (containing all selected "red" edges and corresponding to A_R) and $A_X{}^g$ (containing all selected "green" edges and corresponding to A_H) must be *closed*. That is:
if (ij)$\in A_X$ and ik $\geq_i$ ij then (ik) or (ki)$\in A_X$ (unless of course ik already is in A). All this is described exactly in Definition 15 and Fig. 3.6 of Section 3.2.

As indicated in the previous section CONFLICT (see Fig. 3.5) is a special case of the game CONFLICT$^+$ (see Fig. 3.6), which is important because of its relation to PREFIX (see Fig. 3.4). The inputs of CONFLICT$^+$ are slightly more general constructs, (i.e., ordered mixed multigraphs), instead of mixed graphs. They are motivated from conflict graphs and realizable assignments of directions to their edges.

From Definition 10 Section 2.1 and Lemma 1 section 2.2, we have that a prefix $\langle T,\alpha\rangle$ uniquely determines an ordered mixed multigraph. This is because, given $\langle T,\alpha\rangle$ we can construct $G^{\alpha}(T)=(V,E,A,\{\geq_i\})$, which is the conflict graph ($G(T)$), with some conflicts resolved (A), some conflicts unresolved (E), and the transaction orderings on the *unresolved* conflicts. The assignment of directions A is closed (with respect to the conflict graph $G(T)$) and moreover if C=SR it has no directed cycles (see Theorem 2).

Lemma 2: Given a prefix $\langle T,\alpha\rangle$ in PR(SR) and a nonnegative integer b, then the minimax length of PREFIX($\langle T,\alpha\rangle$) equals the minimax length of CONFLICT$^+$($G^\alpha(T)$).

Proof: Let us recall the following facts:

(a) An assignment of directions (A_Z) to undirected edges (Z) of the conflict graph $G(T)$ is realizable by a prefix iff:
(i) A_Z is closed (with respect to $G(T)$)
(ii) A_Z has no directed cycle ($i_1 i_2 \ldots i_n i_1$) s.t.: $i_1 i_2 \geq_{i2} i_2 i_3, \ldots, i_n i_1 \geq_{i1} i_1 i_2$.

(b) Consider two realizable assignments A,A' of directions to edges of a conflict graph $G(T)$ and let $\langle T,\alpha\rangle$ be a prefix realizing A. It is easy to see that if $A \subseteq A'$ then $A' \setminus A$ is closed (with respect to $G^\alpha(T)$).

(c) Also recall that continuations $\langle T,\beta\rangle$ of $\langle T,\alpha\rangle$ in PREFIX, with projections α_i i=1,2 have properties:
$\langle T,\alpha\rangle$ realizes A, A has no cycles
$\langle T,\beta\rangle \notin$ PR(SR), $\langle T,\beta\rangle$ realizes A', A' has a cycle
$\langle T,\alpha_i\rangle \in$ PR(SR), $\langle T,\alpha_i\rangle$ realizes A_i, A_i has no cycle i=1,2.
We have that, $A' \setminus A$, $A_1 \setminus A$, $A_2 \setminus A$ are closed (with respect to $G^\alpha(T)$). Moreover if 1 is the "red" site and 2 the "green" site and $A_X = A' \setminus A$ then we have $A_X{}^r = A_1 \setminus A$, $A_X{}^g = A_2 \setminus A$.

To prove the lemma we use induction on j, where j=|actions in T and not in α|. For j=0 and any b the lemma is true, since no move is possible (all conflicts are resolved). We will assume the lemma is true for all b and all j, $0 \leq j \leq j^*-1$ and prove it true for j^*. For every move in one game we will exhibit a move in the other, leading to assignments realizable only by stictly larger prefixes.

"only if" from the discussion above a move in PREFIX corresponds to a move in CONFLICT$^+$ and no matter what the choice of Player II is the resulting assignment of directions to the conflict graph $G(T)$ is strictly larger than A and realizable.

"if" A move in CONFLICT$^+$ produces assignments A_X, $A_X{}^r$, $A_X{}^g$. Since these are closed (with respect to $G^\alpha(T)$) and the existing directed part of the board A is closed (with respect to $G(T)$) we have that $A_X \cup A$, $A_X{}^r \cup A$, $A_X{}^g \cup A$ are closed (with

respect to $G(T)$).

We will show that $A_X \cup A$ is realizable by a $\langle T,\beta\rangle$, which is a continuation of $\langle T,\alpha\rangle$. Using Lemma 1, Section 2.2 all that remains to be proven is that $A_X \cup A$ has no directed cycles of form (ii) above. It is easily seen that such a cycle would be completely contained (because of the closure property) in either $A_X{}^r \cup A$ or $A_X{}^g \cup A$. But since $A_X{}^r \cup A$, $A_X{}^g \cup A$, must be acyclic such a cycle cannot exist. Thus $A_X \cup A$ is realizable, in fact using the construction of Lemma 1, Section 2.2 we can choose $\langle T,\beta\rangle$ to be a continuation of $\langle T,\alpha\rangle$. Then it is easy to verify that $A_X{}^r \cup A$, $A_X{}^g \cup A$ are the assignments determined by the projections of $\langle T,\beta\rangle$ (which are strictly larger than A).

Thus when CONFLICT$^+$ has a move PREFIX has one also.□

We will now prove that CONFLICT$^+$(G) is ***PSPACE-Complete***, even if the directed part of G is empty.

Lemma 3: Given an ordered graph $G=(V,E,\emptyset,\{\geq_i\})$ and a nonnegative integer b, determining whether the minimax length of $\text{CONFLICT}^+(G)$ is greater than b is ***PSPACE-Complete.***

Proof: For an arbitrary instance I_n of QBF we can construct the mixed graph $G'(I_n)$ using the following subgraphs.

(a) For x_i, $i=1,3,...,n-1$ $\exists$-subgraphs of Fig. 4.2(b). These are similar to those employed in Lemma 1, with additional nodes A_i,B_i,M_i,N_i and their edges.

(b) For x_i, $i=2,4,...,n$ $\forall$-subgraphs as in (b) of Lemma 1.

(c) For every clause in $F(x_1,...,x_n)$ clause-subgraphs as in (c) of Lemma 1.

(d) The connections are as in (d) of Lemma 1, with the added edges:
directed (A_iA_{ii+2}), $(A_{ii+2}B_{i+2})$ $i=1,3,...,n-3$
directed (A_jA_{jj+1}), $(A_{jj+1}F_{j+1})$ $j=1,3,...,n-1$
undirected "red" A_iB_{i+2}, $i=1,3,...,n-3$, A_jF_{j+1} $j=1,3,...,n-1$.

An example is exhibited in Fig. 4.4. Using $G'(I_n)$ we can construct the following ordered graph $G(I_n)=(V,E,\emptyset,\{\geq_i\})$. Assume I_n has n variables and m clauses:

V: The vertex set of $G'(I_n)$ with an additional vertex for every directed edge in $G'(I_n)$, which has $K_n=10n+35m-2$ directed edges. $|V|=18n+64m-2$.

E: These are the undirected edges of $G'(I_n)$, partitioned into "red" and "green" as in $G'(I_n)$ moreover we replace every directed edge (RQ) of $G'(I_n)$ (see Fig. 4.5(a)) with a triangle (see Fig. 4.5(b)). Thus $G(I_n)$ has no directed edges. It is a graph partitioned into "red" and "green" and has $23n+n/2+91m-5$ "red" and $11n+35m-1$ "green" edges.

$\{\geq_i\}$: To every edge incident at a node i we assign a number. We use the rules of Fig. 4.6 and Fig. 4.5(c). The ordering $\geq_i$ is the strict (no two different elements are equal) total ordering imposed by these numbers at i.
For the kth triangle PQR $1\leq k\leq K_n$, which replaces a directed edge (RQ) we assign:

at P	$PQ\leftarrow 1+kK_n$	$PR\leftarrow 2+kK_n$
at Q	$QP\leftarrow 1+kK_n$	$QR\leftarrow 2+kK_n$
at R	$RP\leftarrow 1+kK_n$	$RQ\leftarrow 2+kK_n$

For the undirected edges of $G'(I_n)$ we use the numbers 1,2,3 as in Fig. 4.6. Note:

at A_i	$A_iB_i\geq A_iF_{i+1}\geq A_iB_{i+2}$	$i=1,3,...,n-1$	(the last for $i\neq n-1$)
at F_{i+1}	$F_{i+1}A_i\geq F_{i+1}E_{i+1}$	$i=1,3,...,n-1$	
at B_{i+2}	$B_{i+2}A_i\geq B_{i+2}A_{i+2}$	$i=1,3,...,n-3$.	

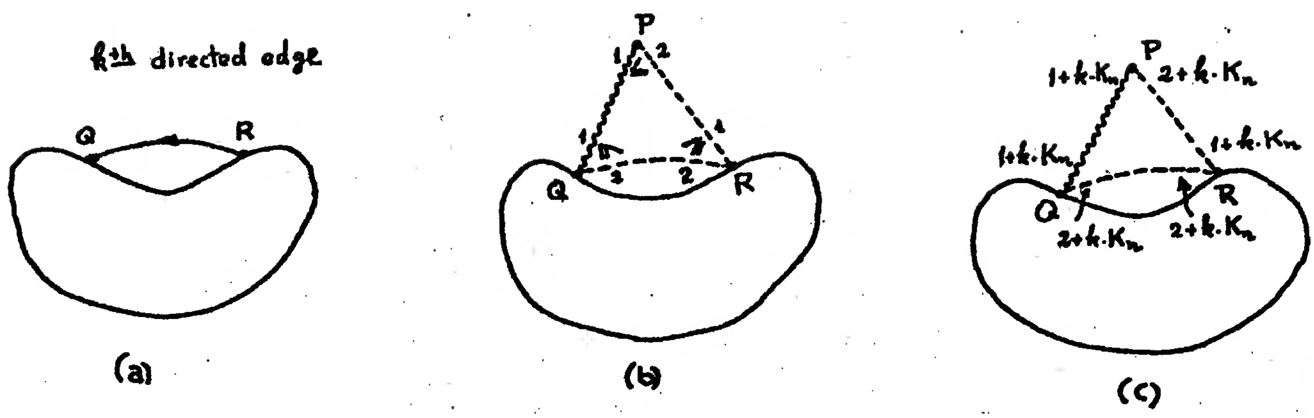


Figure 4.5

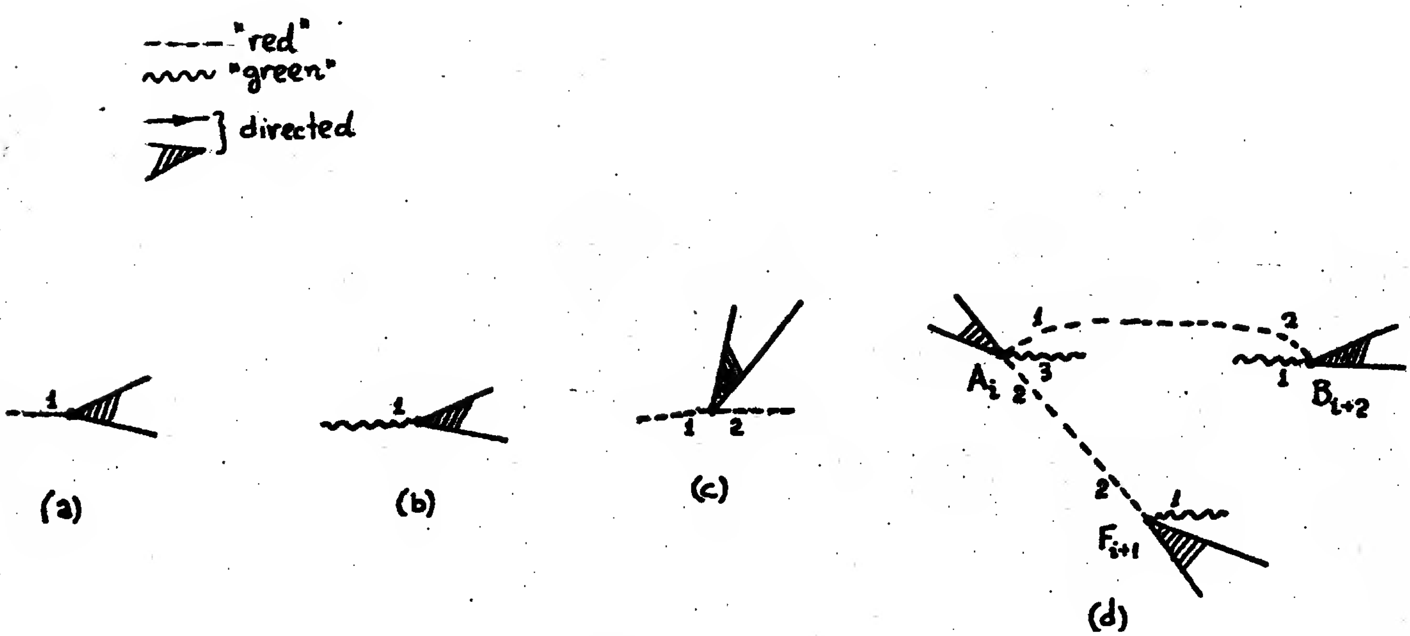


Figure 4.6

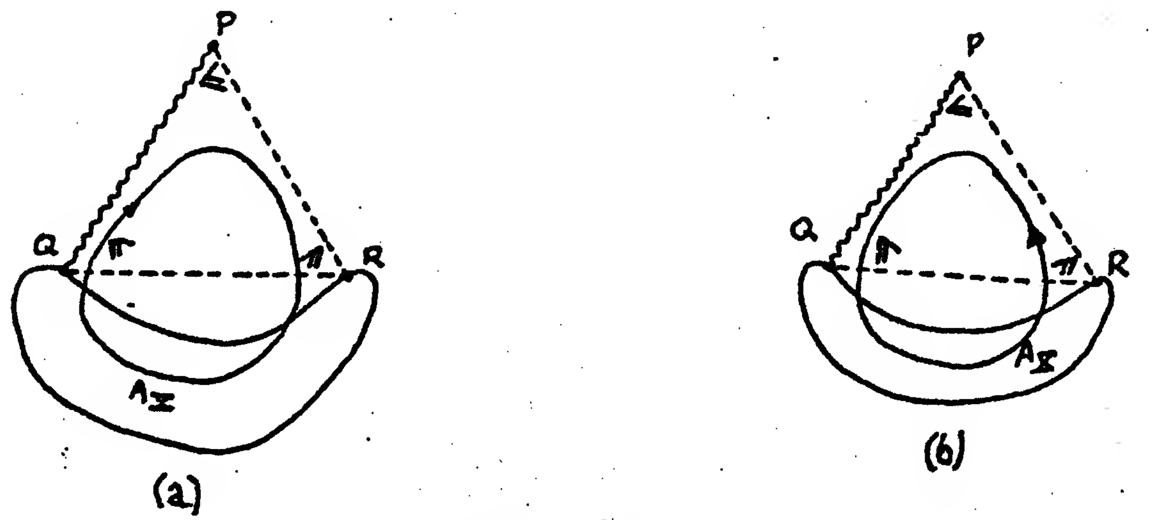


Figure 4.7

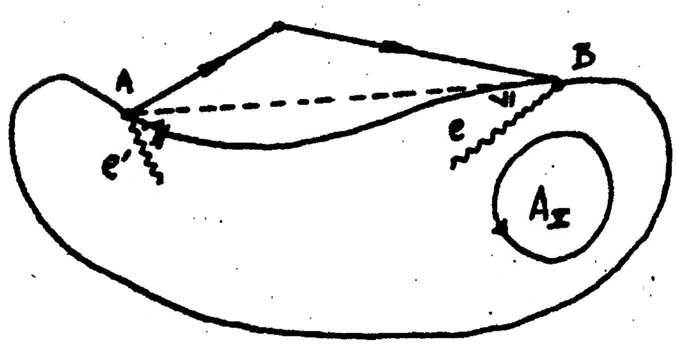


Figure 4.8

We will prove that I_n is true iff CONFLICT$^+$($G(I_n)$) can last more than 2z-2 moves, where $z=11n+35m-1$ (the number of "green" edges).

"only if" Assume that I_n is true. We will describe a strategy that will enable Player I to make z moves (and the game to last 2z moves).

First Player I plays on all the triangles, that we substituted for directed edges of $G'(I_n)$. At his kth move he plays in the K_n-k+1st triangle $1\leq k\leq K_n$ (PQR in Fig. 4.5). The first move is:
$A_X=\{(QP),(PR),(RQ)\}$
$A_X{}^r=\{(PR),(RQ)\}$
$A_X{}^g=\{(QP),(RQ)\}$
These are closed assignments (Def. 15, Section 3.2), with respect to the position of the board. Moreover if (A) are the directed edges on the board before the kth move $A_X \cup A$ has a cycle, while $A_X{}^r \cup A$, $A_X{}^g \cup A$ do not. No matter what Player II's choices will be RQ becomes the directed (RQ) in the new A. By induction Player I can play similarly on all triangles. Note that when Player I has played in all K_n triangles PQR, all (RQ)'s are in the directed part of the board and the directions of the other edges of the triangles have been fixed. Thus without loss of generality we can assume all directions on the triangles as being in A and exclude them from our further arguments about closed assignments.

Now Player I will make n moves alternating between ∃- and ∀-subgraphs (which correspond to the variables of I_n x_i, $i=1,\ldots,n$), from the subgraph of x_1 to the subgraph for x_n. Recall that QBF I_n can be viewed as the instance of a game between two players (the ∃-player and the ∀-player), where the ∃-player has a winning strategy. Player I will pattern his strategy on the winning strategy of the ∃-player of the QBF game (for moves $i+K_n$, $1\leq i\leq n$).

The $i+K_n$th move of Player I ($1\leq i\leq n$) is:

(a) If $i=1,3,\ldots,n-1$ and the ∃-player makes $x_i=x^*_i=1$ (based on the values x^*_j that have been assigned for $1\leq j\leq i$) then:
$A_X=\{(T_iD_i),(D_iM_i),(B_iA_i)$, and $(A_{i-2}B_i)$ if $i>1\}$
$A_X{}^r=\{(T_iD_i),(D_iM_i)$, and $(A_{i-2}B_i)$ if $i>1\}$
$A_X{}^g=\{(B_iA_i)$, and $(A_{i-2}B_i)$ if $i>1\}$
It is easy to check tht if the board position has directed edges A, these assignments

are closed. Also $A_X \cup A$ has cycle $(T_iD_iM_iB_iA_iT_i)$ and $A_X{}^r \cup A$, $A_X{}^g \cup A$ do not have any cycle ($A_{i-2}B_i$'s direction had been fixed to $(A_{i-2}B_i)$ anyway). No matter what the response of Player II is to this move, the path $(S_iT_iD_iS_{i+1})$ and the new directed part of the board form no directed cycles.
If the $\exists$-player makes $x_i=0$ we use the symmetric cycle $(F_iE_iN_iB_iA_iF_i)$.

(b) If $i=2,4,...,n$ then Player I uses cycle $(T_iD_iF_iE_iT_i)$.
$A_X=\{(T_iD_i),(F_iE_i),(A_{i-1}F_i)\}$
$A_X{}^r=\{(T_iD_i),(A_{i-1}F_i)\}$
$A_X{}^g=\{(F_iE_i),(A_{i-1}F_i)\}$
Again it is easy to see that the move is legal. But now Player II's response is significant. A choice "red" would correspond to the $\forall$-player assigning $x_i=x^*_i=1$ and would fix directions to (T_iD_i) and (E_iF_i). Then $(S_iT_iD_iS_{i+1})$ only forms no cycles with the new directed part of the board. A choice "green" would be symmetric (i.e. $x_i=x^*_i=0$ and only $(S_iF_iE_iS_{i+1})$ forms no cycles with the new directed part of the board).

We have now reached the zth $(z=n+K_n+1)$ move of Player I, and the $\exists$-player has won his QBF game on I_n using assignment $\{x^*\}$. Thus the derived assignment $\{u(x^*)\}$ to the literals makes every clause of the formula of I_n true. We can use the same move as was the last move in Lemma 1 and trivially check that it is legal.

"if" If I_n is false we will prove, that although 2z-2 moves are possible, 2z moves are not, in CONFLICT$^+(G(I_n))$. In this case $\neg I_n$ is true and the $\forall$-player has a winning strategy in the QBF game. We will pattern the strategy of Player II on this strategy of the $\forall$-player.

Suppose that CONFLICT$^+(G(I_n))$ can last 2z moves. It is easy to see, that every move of Player I must contain exactly one "green" edge, whose direction has not been fixed by previous moves, (observation (0) before Lemma 1). So we can view sequences of z legal moves by Player I as permutations of the z "green" edges and name every move by the "green" edge, whose direction it fixes.

(a) First let us look at legal PQ-moves, that is moves whose "green" unfixed edge belongs to a triangle. If this move (A_X) produces a cycle as in Fig. 4.7(a) we can infer the following: The edge (RQ) must belong to $A_X{}^r \cup A$ and $A_X{}^g \cup A$. This is because $A_X{}^r \cup A$ must contain a directed path (P...Q) and $QR \geq_Q QP$. (Recall that

QP is the only "green" edge without a fixed direction in A_X). Thus no matter what the response of Player II is to such a PQ-move the edge (RQ) becomes part of A. On the other hand a PQ-move producing a cycle (A_X) as in Fig. 4.7(b) is never legal. This is because $A_X^g \cup A$ must contain {(PQ),(QR),(RP)} a cycle. The existence of a path (Q...P) in $A_X^r \cup A$ and the fact that $RQ \geq_R PR \geq_P QP$ force this situation. Thus PQ-moves fix the direction of QR to (RQ). Finally if Player I were ever to use a QR in the direction (QR), in some other e-move (e a "green" unfixed edge), then a response of "red" by Player II would consume two "green" edges (i.e., e and PQ). *Therefore Player I should regard edges RQ as directed (RQ), in order to be able to play z times.*

(b) Let us examine the A_iB_i-moves $i=1,3,...,n-1$ and F_iE_i-moves $i=2,4,...,n$. Since the directed edges of $G'(I_n)$ have to be respected, we can only have $(B_iA_i) \in A_X$ and $(F_iE_i) \in A_X$ for legal assignments in these moves. This is because $A_X \cup A$ must contain a cycle and all other edges incident at A_i (respectively F_i) have fixed outgoing (respectively ingoing) directions. Now we can justify the construction in Fig. 4.6(d) and 4.8. If $(B_iA_i) \in A_X$ from the $\geq_{Bi}$ order we have that $(A_{i-2}B_i) \in A_X \cup A$ (e.g. the direction of $A_{i-2}B_i$ is fixed to $(A_{i-2}B_i)$ because of the directed path $(A_{i-2}A_{i-2,i}B_i)$ in $G'(I_n)$). From the $\geq_{Ai-2}$ order we have that $(B_{i-2}A_{i-2})$ or $(A_{i-2}B_{i-2}) \in A_X \cup A$. Similarly if $(F_iE_i) \in A_X$ then $(B_{i-1}A_{i-1})$ or $(A_{i-1}B_{i-1}) \in A_X \cup A$. *We have established that the A_iB_i-move must precede the $A_{i+2}B_{i+2}$- and $F_{i+1}E_{i+1}$-moves $i=1,3,...,n-1$.*

(c) Finally let us examine the $C_{m+1}S_1$-move. For this move we need a simple path $(S_1...C_{m+1})$ that respects directed edges in $G'(I_n)$, can contain no backedges of $G'(I_n)$ (similarly to (3) of Lemma 1), and has to pass through S_n and S_{n+1} (the last ∀-subgraph). If the F_nE_n-move has not been played yet the use of either (T_nD_n) or (F_nE_n) by the $(S_1...C_{m+1})$ path would fix the direction of F_nE_n. *Thus the C_mS_1-move has to follow all the A_iB_i- and F_iE_i-moves $i=1,2,...,n$.*

We will now show that Player II can force Player I in a game, which simulates a QBF(I_n) game, where Player I is the ∃-player, Player II is the ∀-player and moreover has a winning strategy. Player I chooses the values of x_i $i=1,3,...,n-1$ and II the values of x_i $i=2,4,...,n$. Player I determines when Player II makes his choices (as long as x_i precedes x_{i+1} $i=1,3,...,n-1$). Thus the best Player I can do is assign a value to x_1, force II to assign a value to x_2, assign a value to x_3, etc. Let us describe how these assignments take place.

(1) *The A_iB_i-move assigns a value to x_i $i=1,3,\ldots,n-1$.* The only possible choices for A_X are cycles $(B_iA_iT_iD_iM_iB_i)$ for $x^*_i=1$ or cycles $(B_iA_iF_iE_iN_iB_i)$ for $x^*_i=0$. This is because directed edges in $G'(I_n)$ must be respected, and for $x^*_i=1$ we have the following ($x^*_i=0$ is symmetric):

$(B_iA_iB_{i+2}A_{i+2}\ldots)$ would use up $B_{i+2}A_{i+2}$.

$(B_iA_iT_iD_iF_iE_i\ldots)$ would introduce a cycle in $A_X{}^r \cup A$.

$(B_iA_iT_iD_iS_{i+1}\ldots)$ would fix the direction of $F_{i+1}E_{i+1}$.

The strategy of Player II will be to always play "red", fixing the directions of T_iD_i, F_iE_i and making vertex A_i inaccessible from S_i.

(2) The F_iE_i-move assigns a value to x_i $i=2,4,\ldots,n$. The arguments are exactly as in the $\forall$-subgraphs of Lemma 1. Player II's choice fixes the direction of F_iE_i, thereby making x^*_i 1 or 0 and allowing a unique path from S_i to S_{i+1} as in Lemma 1. Player II assigns values to the x^*_i's according to the winning strategy of the $\forall$-player (recall that I_n is false and thus the $\forall$-player has a winning strategy).

As a result of all this analysis we see that when it is time for the $C_{m+1}S_1$-move, Player II has forced $F(x^*_1,\ldots,x^*_n)$ to be false, and constrained $(S_1\ldots C_{m+1})$ to a unique path through the $\exists$- and $\forall$-subgraphs (e.g., the labels on the path are $\{x^*\}$ exactly as in Lemma 1).

Thus the arguments of Lemma 1 apply to show that the C_mS_1-move cannot be legal and $\text{CONFLICT}^+(G(I_n))$ cannot last $2z$ moves. □

We have now practically completed the proof of Theorem 4.

Proof of Theorem 4: In Lemma 1 we have proven that CONFLICT(G) is Π_2^P*-hard.* Using this lemma we have shown, in Lemma 3, that CONFLICT$^+$(G) is ***PSPACE-Complete*** for G an ordered graph (no directed edges). In Lemma 2 we have shown the equivalence of PREFIX($\langle T,\emptyset\rangle$) and CONFLICT$^+$(G($T$)). In order to complete Theorem 4 all we have to do is argue that the ordered graph in the reduction of Lemma 3 is a conflict graph for some T:

In fact $G(I_n)=(V,E,\emptyset,\{\geq_i\})=G(T)$ because,
V: every vertex i corresponds to a transaction T_i.
E: every edge e=ij corresponds to transactions T_i and T_j updating a uniquely defined variable x_e. If e is "red" x_e is stored at site 1, if e is "green" x_e is stored at site 2.
$\{\geq_i\}$: All orders are strict total orders, because every edge ij is assigned a different number at i, thus all vertices are realizable by transactions.

Thus we have shown PREFIX($\langle T,\emptyset\rangle$) to be ***PSPACE-Complete.***□

The question, whether PREFIX($\langle T,\alpha\rangle$) can last more than b moves, has several interesting subcases.

For $\langle T,\alpha\rangle$:
(1) $G^\alpha(T)$ is a graph and $\{\geq_i\}$ are strict, (e.g., every transaction updates a variable only once. Two transactions never share more than one variable. Three transactions do not share a variable).
(2) $\alpha=\emptyset$ (e.g., there are no directed edges or all conflicts are unresolved)
(3) The transactions in T contain no cross-edges (e.g., each $\geq_i$ consists of two total orders one "red" and one "green". The "red" and "green" edges are incomparable. This actually means that there are no transaction defined messages).
(4) The $\{\geq_i\}$ are of fixed size (e.g., no more than L actions per transaction).

For b whether it is arbitrary or 0.

These cases with their complexities are exhibited in Table 1.

conditions ($\langle T,\alpha\rangle$)	Complexity	Complexity (b=0)
(1)&(2)	*PSPACE-Complete* Theorem 4	in *NP*
(1)&(2)&(4) L=6	*PSPACE-Complete* Corollary 4.1	in *NP*
(1)&(2)&(4) L=4	Π_2^P-*hard* Corollary 4.1	in *NP*
(1)&(3)&(4) L=6	*PSPACE-Complete* Corollary 4.3	*NP-Complete* Corollary 4.2
(2)&(3)	in *PSPACE*	in *P* Corollary 3.3

Table 1: Is minimax length of PREFIX($\langle T,\alpha\rangle$) greater than b ?

<u>Corollary 4.1</u>: Whether the minimax length of PREFIX(<T,∅>) is greater than b is ***PSPACE-Complete***, even if the degree of the graph G(T) is less or equal to 6. It is $\Pi_2{}^P$***-hard*** even if the degree of G(T) is less or equal to 4.

<u>Proof</u>: We will slightly modify the gadgets of Lemma 3 without changing the validity of its arguments. We replace clause-subgraphs with Fig. 4.9a, ∀-subgraphs with Fig. 4.9b and ∃-subgraphs with Fig. 4.9c. Let us for the moment ignore the nodes A_i,B_i i=1,3,...,n-1. The construction gives us (by Lemma 1) that our decision problem is $\Pi_2{}^P$***-hard.*** Moreover the only configurations at nodes are those of Fig. 4.10, thus our transactions need never have more than 4 actions. If on the other hand we add in nodes A_i and B_i and connect A_i,B_{i+2},F_{i+1} using the subgraph of Fig. 4.11 then the arguments of Lemma 3 are still valid. The only difference from Lemma 3 is that $A_i$$B_i$-moves must precede the moves in the triangles corresponding to ($A_i$$B_{i+2}$) and ($A_i$$F_{i+1}$). We can thus show that our decision problem is ***PSPACE-Complete*** even if transactions are restricted to 6 actions.

Therefore [<T,∅>∈ M_c(b)?] is ***PSPACE-Complete*** even if transaction systems are very restricted.□

Consider the following combinatorial problem, which is in ***NP***.

PATH(G,s,t)
<u>Input</u>: A mixed graph G=(V,E,A) (V=set of vertices, E=set of undirected edges, A=set of directed edges) and two distinguished nodes s and t.
<u>Output</u>: Is there an assignment (A_E) of directions to the edges in E, such that the digraph (V,A_E∪A) is acyclic, and contains a directed path from s to t?

Note that, if A is acyclic, there is always an $A_E{}^*$ such that (V,$A_E{}^*$∪A) is acyclic. Also it is easy to determine in the mixed graph G=(V,E,A) if t is reachable from s. But both conditions simultaneously are hard to decide.

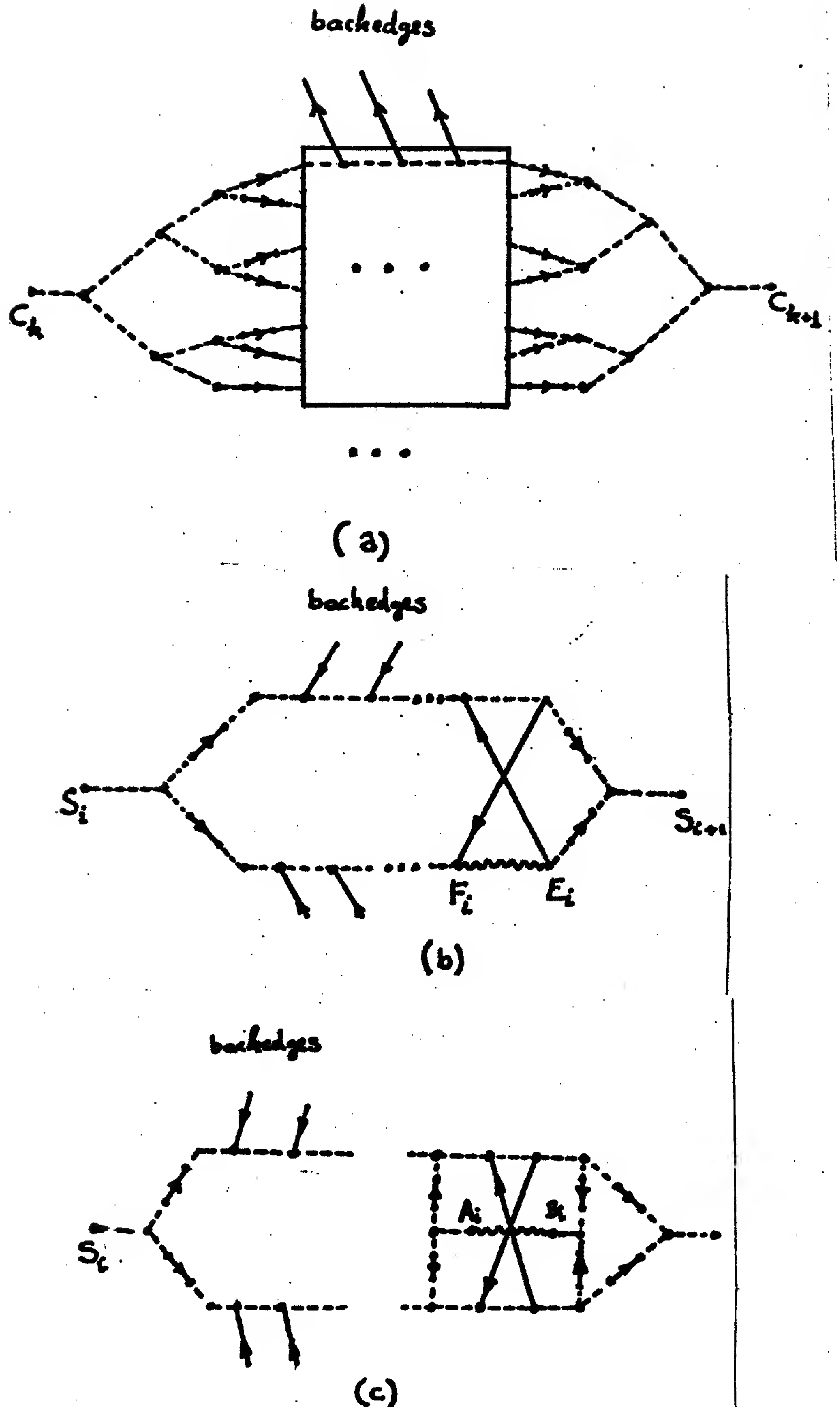

backedges
C_k
C_{k+1}
(a)
backedges
S_i
S_{i+1}
F_i
E_i
(b)
backedges
S_i
A_i
B_i
(c)

Figure 4.9

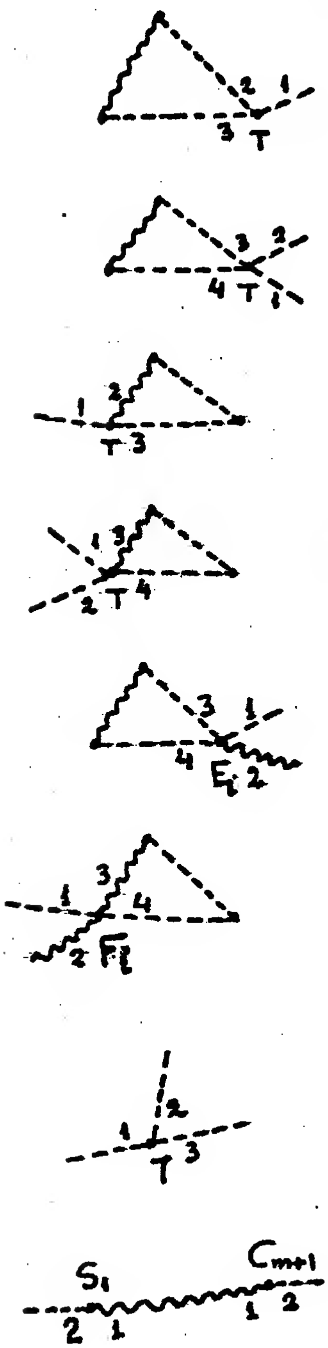


Figure 4.10

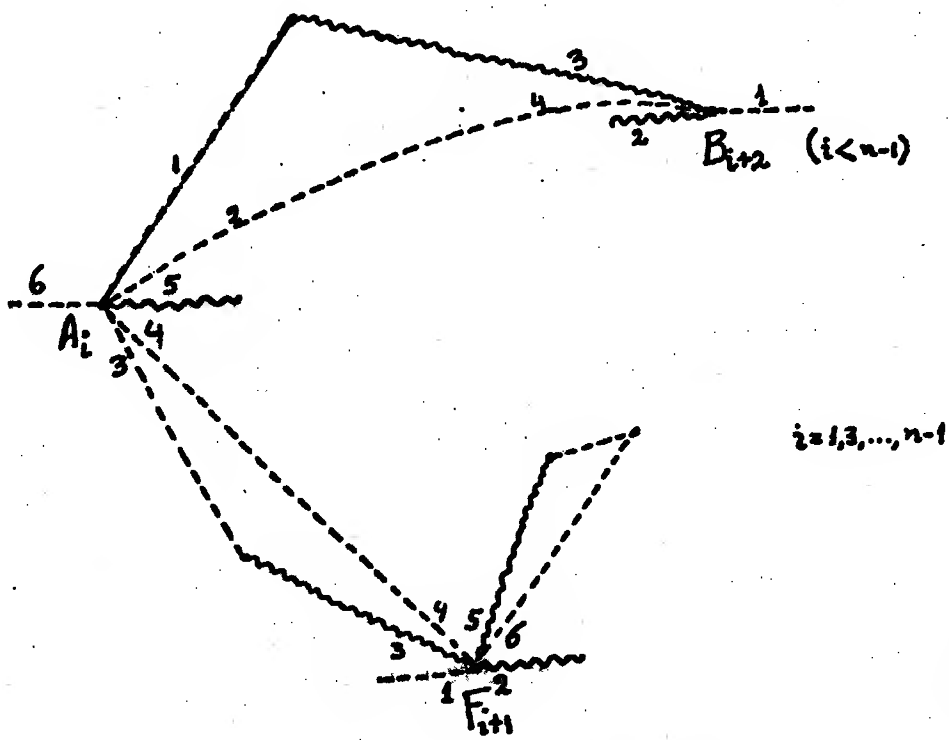


Figure 4.11

Corollary 4.2: PATH(G,s,t) is *NP-Complete*, even if G has at most 2 undirected edges incident at a node and at most 1 directed edge incident at a node.

Proof: Consider Lemma 1, where all edges F_iE_i become "red". Then the frst player chooses the values for all variables, and our QBF game becomes the satisfiability problem. Finding a proper path $(S_1...C_{m+1})$ would answer PATH($G(I_n),S_1,C_{m+1}$). This and the refinements of Corollary 4.1 prove the Corollary.□

Corollary 4.3: Whether the minimax length of PREFIX($\langle T,\alpha\rangle$) is greater than b is *PSPACE-Complete* (for b arbitrary) and *NP-Complete* (for b=0). This is true even if the transactions in T have no cross-edges and a fixed number of actions.

Proof: Another way of stating Corollary 4.2 is that the decision question [$\langle T,\alpha\rangle \in M_c(0)$?] is co-NP-Complete. The analysis that follows (for b arbitrary) also applies to this case. therefore determining if PREFIX($\langle T,\alpha\rangle$) can last more than 0 moves is *NP-Complete*.

In Lemma 3 we totally ordered all edges incident at a node, by assigning numbers to them. Thus the transaction system realizing the $\{\geq_i\}$ of $G(I_n)$ had to have cross-edges. In fact cross-edges are the only way we know of forcing the creation of desired directed edges.

Given an instance of QBF(I_n) we can construct the ordered mixed graph $G''(I_n)$ as follows (recall the mixed graph $G'(I_n)$ and the ordered graph $G(I_n)$ of Lemma 3):

$G''(I_n)=(V,E,A,\{\geq_i\})$ where:
$(V,E,A)=G'(I_n)$, with one exception. The edges A_iB_{i+2} (i=1,3,...,n-3) and A_iF_{i+1} (i=1,3,...,n-1) are "green" and not "red".
$\{\geq_i\}$ are those implied by the orders of $G(I_n)$ of Lemma 3.

We can prove that I_n is true iff CONFLICT$^+$($G''(I_n)$) can last more than n moves. The argument that is needed to prove equivalence is identical with that of Lemma 3. Note that $G''(I_n)$ has 2n "green" edges of which n-1 have fixed directions.

It is easy to see that a prefix $\langle T,\alpha\rangle$ can be constructed, from a transaction system without cross-edges, such that $G^\alpha(T)=G''(I_n)$. ($G^\alpha(T)$ is $G(T)$ with the resolved

conflicts). Thus by Lemma 2, we complete the proof of Corollary 4.3. By using the gadgets of Fig. 4.9 we can restrict the transaction systems to sets of transactions with at most 6 actions (e.g., the nodes A_i have two "green", two "red" and two directed outgoing edges. "Green" and "red" edges at the same node are incomparable).

This proves that the decision question [$\langle T,\alpha\rangle \in M_c(b)$?] is ***PSPACE-Complete*** even for Ts without cross-edges and with a fixed number of actions per transaction.□

From this analysis of special cases we see that two sets of constraints give us equal power:
{(1)&(2)&(4) L=6} and {(1)&(3)&(4) L=6}

Let us now examine the final special case, namely b=0. Since we fix b we cannot use the equivalence above. From Corollary 4.2 we have that if $\alpha \neq \emptyset$ and if T has no cross-edges the problem is ***NP-Complete***. From Corollary 3.3 if T has no cross-edges and $\alpha = \emptyset$ the problem is in ***P***.

We have left *open* two interesting problems:

(a) Given T without cross-edges and b≥0, is the minimax length of PREFIX($\langle T,\emptyset\rangle$) greater than b moves? We conjecture this problem is ***PSPACE-Complete***.

(b) Given T can PREFIX($\langle T,\emptyset\rangle$) last more than 0 moves? This problem is in ***NP*** and we conjecture it is also in ***P***.

4.2 The Efficiency of Communication-Optimal Schedulers

In the previous section we have analysed the complexity of various cases of PREFIX, or equivalently examined various cases of the decision problem [$\langle T,\alpha\rangle \notin M_c$(b)?]

In Section 3.1 we described a programming system in which we can express all distributed schedulers. These schedulers consist of two processes, one at each site (Q_1,Q_2) and realize SR (Definition 12, Section 3.1). That is an input history h∈H can lead to many possible computation paths. By executing the instructions on such a path the scheduler outputs a history in C. For each path the output is in C, moreover if h∈C and the delays of all messages are 0 the output must be h. We call the scheduler polynomial time bounded if the number of instructions the processes execute is bounded by a polynomial in n (for all possible paths). The size of the input is measured by n, which is the number of actions in T.

Corollary 4.4: Unless *NP=PSPACE*, there is no communication-optimal scheduler, which realizes SR and is polynomial time bounded. This is true even if each transaction is restricted to be a sequence of six updates.

Proof: Suppose such a scheduler Q existed. We know that [$\langle T,\emptyset\rangle \notin M_c$(b)?] is *PSPACE-Complete* (even for restricted transaction systems, Corollary 4.1). We will prove that there exists a nondeterministic polynomial time bounded decision procedure for this problem. This would imply that *NP=PSPACE*, an unlikely fact.

Given T and b≥0 we do the following:

(1) guess a history h=$\langle T,\pi\rangle \in$ SR (this can be easily checked)

(2) simulate the operation of Q on this history

(3) whenever a message is sent we guess its delay and in general guess a computation path of Q.

(4) keep count (with m) of the number of messages sent

(5) if m>b then say yes else say no

If [$\langle T,\emptyset\rangle \notin M_c(b)$?] is true there will exist an input h and a computation path of Q, where more than b instructions are executed. We can guess the input and the computation path with a polynomial number of guesses, this is because the size of h is O(n) and all paths are polynomial bounded. If m>b that means that all schedulers have to use more than b messages for inputs from the transaction system T. This is obviously a nondeterministic polynomial time bounded algorithm for our problem.□

Similar results would hold even if we augmented our programming system with the power to consult oracles in the polynomial hierarchy [11] (i.e., the hierarchy would collapse beyond a certain level).

Let us note two *open* problems.

(a) If we assume ***P=PSPACE*** it follows that we can construct efficient schedulers (in both measures). The consequences of ***NP=PSPACE*** on the other hand are unclear.

(b) If the decision problem [$\langle T,\emptyset\rangle \notin M_c(b)$?] is only ***NP-hard*** the arguments of Corollary 4.4 no longer apply.

Our results indicate that a communication optimal scheduler must be computation inefficient. It is still possible to analyze the information in T and design various efficient, communication suboptimal realizations of SR. We will end this section by defining a simple open edge deletion problem. This problem can be used as an upper bound on the minimum number of messages in order to realize SR. Because of its simplicity it is also of independent combinatorial interest.

DMC(G)
Input: An undirected graph G, with edges partitioned into "red" and "green"
Output: Find the minimum number of edges, whose deletion produces a graph with no cycles containing both "red" and "green" edges.

5. The Combinatorics of Locking

The most common technique used for the resolution of conflicts in concurrency control is locking. In this chapter we will extend the elegant analysis of locking described in [39] from the centralized to the distributed case. In the process, the geometric criterion of [39] will be replaced by a simple combinatorial condition (i.e., the strong connectivity of a directed graph).

5.1 Distributed Locking

Let us first present a simple extension of the definitions for locking, which appear in [39]. We will utilize the notions of Distributed Database Design (DDD), transaction, action, history and serializability from Section 2.1, with the following additions:

Definition 16: For the DDD=$\langle G_D, Data, Stored\text{-}at, IC \rangle$, the *Data* is partitioned into **variables** (Var) and **locking variables** (LVar). The function *lock-of*: Var→LVar determines for every variable x, its lock X, (i.e., X is the *lock-of*(x)). The constraint $\wedge_{(X \in LVar)}\ X=0$ is part of the integrity constaints IC.□

We will use x for variable and X for its lock. Note that, as for all *Data*, locking variable X is *stored-at site(X)*. We might have that *site(x)≠site(X)* (e.g., a central site is used for all locks). We might have that X is the lock of x only and *site(x)=site(X)* (e.g., the fully distributed case). Or we could have two variables, which are at the same or different sites, and have the same lock (e.g., primary copy locking). The locks we will be dealing with are stored at a particular site, and are not global variables stored at many sites.

The transactions and histories are partial orders of actions as in Definitions 2 and 4, but we can have more types of actions.

<u>Definition 17</u>: An action is either an update of a variable (in Var) as defined in Def. 3 or a **lock** X or **unlock** X step for some locking variable X (in LVar).
(a) The semantics of "lock X" are, (X:= if X=0 then 1 else *error*)
(b) The semantics of "unlock X" are, (X:= if X=1 then 0 else *error*)
We abreviate "lock X" as Lx and "unlock X" as Ux, where X=*lock-of*(x).□

Note that we are dealing with *exclusive* locks. We will not discuss *shared* locks (e.g., read or intention locks [13])

Let $\mathcal{T}=\{T_1,T_2,\ldots,T_m\}$ denote an (ordinary) transaction system, that is without "lock" or "unlock" steps.

<u>Definition 18</u>: A **locking policy** L is a mapping, which given an (ordinary) transaction system $\mathcal{T}$ transforms it into a **locked transaction system** $L(\mathcal{T})$. The locking policy transforms each T_i of $\mathcal{T}$ into $L(T_i)$ $(i=1,2,\ldots,m)$, by inserting only Lx,Ux steps and precedences between them subject to the following constraints:

(1) The only way to insert Lx or Ux steps, is as a Lx-Ux pair with Lx before and Ux after an update of x, in the partial order of $L(T_i)$. Moreover for each x there is at most one Lx-Ux pair in $L(T_i)$.

(2) For every update of an x in T_i there is a Lx before and an Ux after it in the partial order of $L(T_i)$.□

Note that a locking policy could be nondeterministic (i.e. it could produce many different $L(\mathcal{T})$'s for a given $\mathcal{T}$).

In a locked transaction $L(T_i)$ all actions at the same site are totally ordered, by Def. 3 of transactions. As in the case without locks, a distributed locked transaction represents a set of total orders of its actions (i.e., those that respect its partial order). A new feature for the distributed case is: we can have actions p,q concurrent in T_i and Lx's, Ux's inserted in T_i, with such precedences as to make p an ancestor of q in $L(T_i)$. In other words the locking policy can restrict the parallelism inherent in T_i.

Let h be a history (or a prefix of a history) of $L(\mathcal{T})$. We say that h is *legal* (i.e. preserves the IC of locks in Definition 16) if between any two occurrences of Lx in h there is an occurrence of Ux. We denote this as $h \in M(L(\mathcal{T}))$. Let $L^{-1}(h)$ be the induced subgraph of h if all lock and unlock steps were removed. The set of histories $O(L)=L^{-1}(M(L(\mathcal{T})))$ is called the **output** of the locking policy L and captures the parallelism supported by L.

Definition 19: A locked transaction system $L(\mathcal{T})$ is **safe** if every history in O(L) is serializable. It is **deadlock-free** if for any *legal* prefix α of a history of $L(\mathcal{T})$, there is a suffix ω such that $\alpha.\omega \in M(L(\mathcal{T}))$.□

It is easy to see that if $L(\mathcal{T})$ is safe we can realize $M(L(\mathcal{T})$ using a scheduler, which consists of a simple lock manager and a mechanism for avoiding or breaking deadlocks. The deadlock problem becomes more accute in a distributed environment, where it requires the use of messages [22,23].

As an example of a distributed locking policy consider **two-phase locking** (2PL).

2PL: All lock steps in a distributed locked transaction must precede all unlock steps in the transaction's partial order.

Every total order consistent with a 2PL distributed transaction is a 2PL centralized transaction. Thus we can infer, from the safety of centralized 2PL, its safety for the distributed case. Similar easy generalizations exist for the safe and deadlock-free tree-[30], digraph-[39] or hypergraph-[39] policies, which apply to the structured *Data* case.

An example of a distributed 2PL transaction system is presented in Figure 5.1. This example also shows that O(2PL) (i.e., the set of legal output histories without the locks) is *not* a concurrency control principle as defined in Def. 11 Section 2.1. This is because the ordering of lock, unlock steps introduces cross-edges that were not part of the initial transactions $\mathcal{T}$.

Our main task now will be to generalize the results of [39] towards a characterization of safe systems.

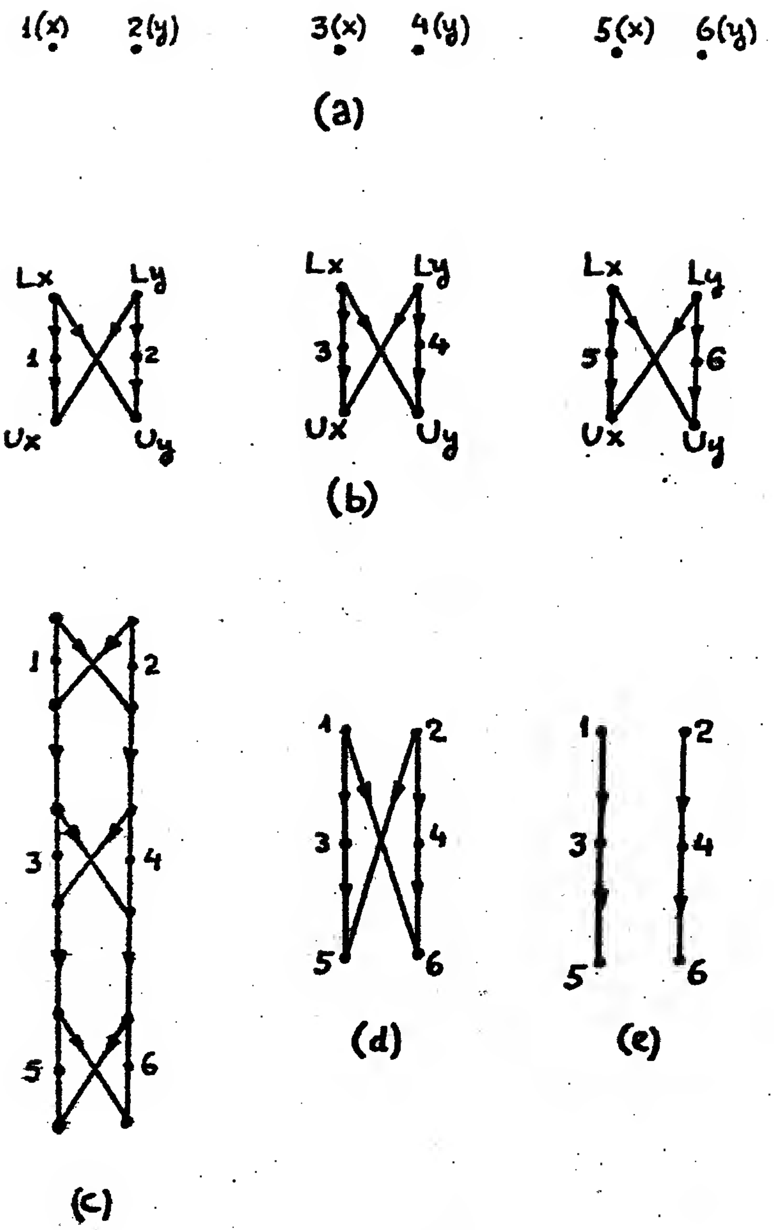


Figure 5.1

(a) T (x,X at site 1 and y,Y at site 2)
(b) L(T) two phase locked
(c) history s where s$\in$ M(L(T))
(d) history h where $h=L^{-1}(s)$, h$\in$ O(L).
(e) h without crossedges is not in O(L)

5.2 The Safety of Distributed Locked Transaction Systems

Let T_i $(i=1,2)$ denote *a pair of locked distributed transactions* and T_i^+ $(i=1,2)$ *a pair of totally ordered locked distributed transactions.* The jth step of T_i^+ is T_{ij}^+ $1 \le j \le m_i$. As noted above $T_i = \{T_i^+ \mid T_i^+ \text{ respects } >_{Ti}\}$ $(i=1,2)$.

Consider a transaction system $\{T_i^+, i=1,2\}$. In the coordinated plane (T_i^+, T_2^+) (see Fig. 5.2) take the two axes to correspond to T_1^+ and T_2^+, and the integer points 1,2, etc. on these axes to correspond to the steps T_{11}^+, T_{12}^+, etc. (respectively T_{21}^+, T_{22}^+, etc.) of the transactions. A point p may represent a possible state of progress made toward the completion of T_1^+ and T_2^+. These transactions will contain properly nested lock-unlock steps. Each variable x such that both T_1^+ and T_2^+ contain a Lx-Ux pair, has the effect of creating a *forbidden region* (a rectangle delimited by the grid lines corresponding to the Lx-Ux steps), the points of which do not represent reachable states (see Fig. 5.2). Adding such *rectangles* to the plane has some consequences. For example, the point u is now reachable, yet not in any rectangle; in contrast, point d is a state of deadlock.

A history, that is totally ordered, has the following geometric image[39]. It is a nondecreasing curve from the point (0,0) to the point (m_2+1, m_1+1), not passing through any other grid point and not through any rectangle (e.g. h in Fig. 5.2). To read the history off any such curve we simply enumerate the grid lines that it intersects. Two totally ordered serial histories are represented by the curves h_1, h_2 in Fig. 5.2.

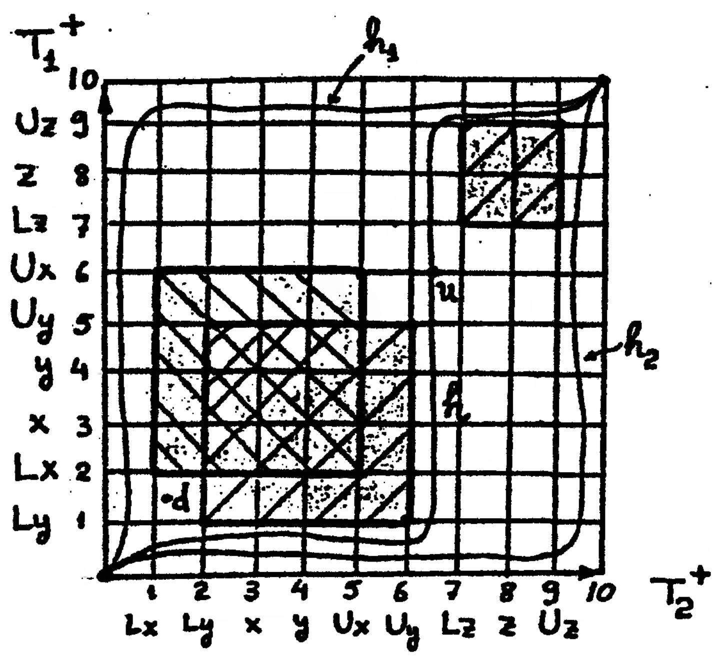


Figure 5.2 The (T_1^+, T_2^+)-plane

From [39] we have the following characterization.

Proposition 2: A history, which is totally ordered, for the transaction system $\{T_i^+,\ i=1,2\}$ is not serializable iff the corresponding curve separates two rectangles.□

No two rectangles touch at a grid point (by our definition of locked transaction systems). In order to study the safety of $\{T_i^+,\ i=1,2\}$ the only actions we have to consider are pairs of Lx-Ux steps, where both T_i^+'s update x. The following Lemma for distributed locked transactions is a direct consequence of Proposition 2, because every nonserializable history corresponds to some set of totally ordered nonserializable histories.

Lemma 1: A distributed locked transaction system $\{T_1,T_2\}$ is safe iff for all pairs T_1^+,T_2^+ there is no curve (corresponding to a history) that separates two rectangles in the (T_1^+,T_2^+)-plane.□

An example of an unsafe system $\{T_1,T_2\}$, where only relevant Lx-Ux steps are given, is provided by Fig. 5.3(a). In Fig. 5.3(b) we have a pair T_1^+,T_2^+ that happens to be safe. In Fig 5.3(c) we have a pair T_1^+,T_2^+ that illustrates why the system is unsafe.

Since there is an exponential number of possible pairs T_1^+,T_2^+ an iterative application of the test of Proposition 2 (which involves an O(nlognloglogn) computation of a "closure" for a geometric region of rectangles [21]) is no longer efficient.

Our contribution will be an efficient combinatorial (as opposed to geometric) test (i.e. sufficient condition) of safety for distributed locked transaction systems. Our combinatorial test (Theorem 5) provides an alternate way of characterizing the centralized problem. It is also a necessary condition of safety (Theorem 6) for centralized transactions and transactions distributed at two sites. For more sites a complete and efficient characterization is an open problem.

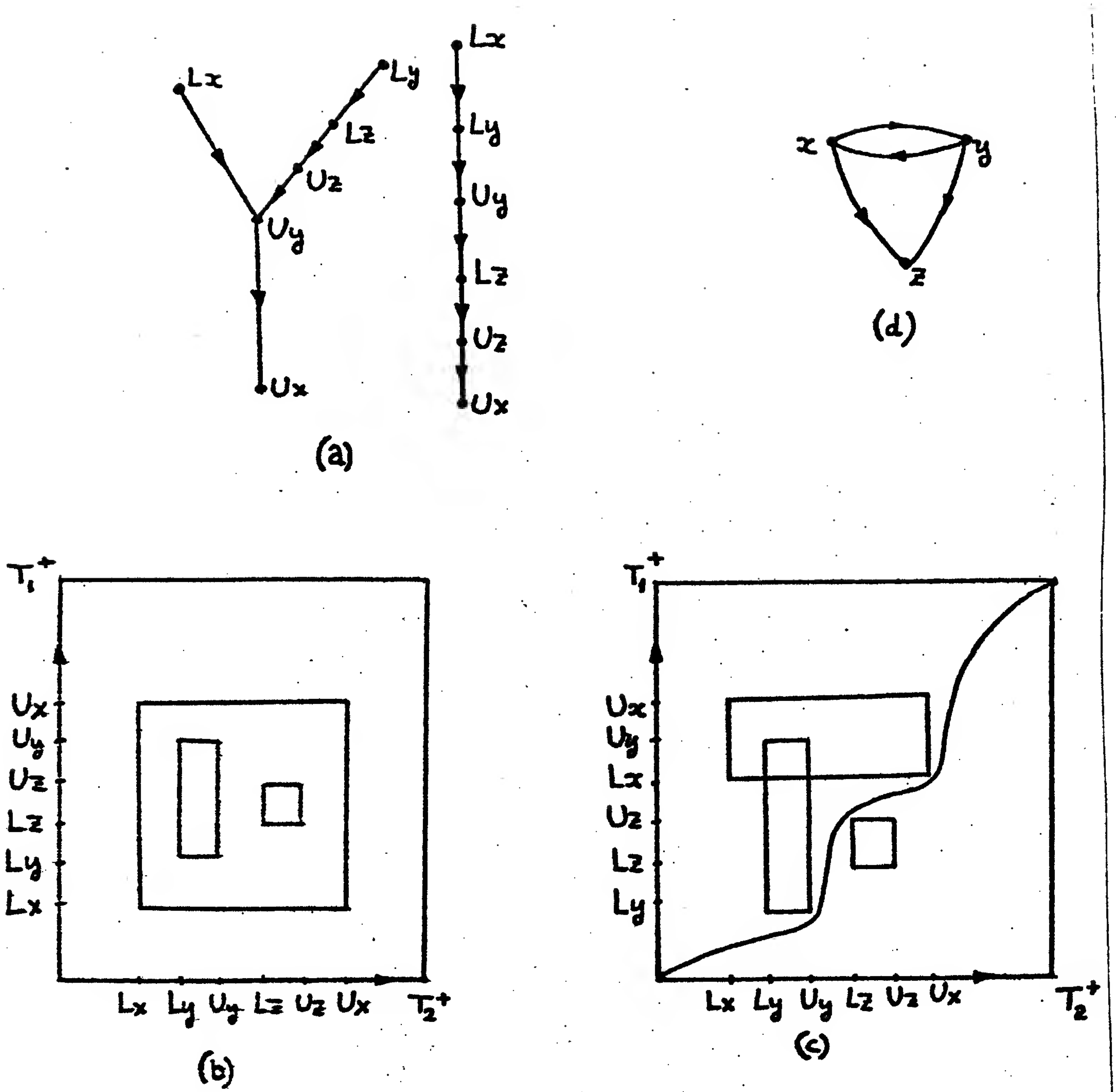


Figure 5.3
(a) Distributed locked transactions (x at site 1 and y,z at site 2)
(b) safe $\{T_1{}^+, T_2{}^+\}$
(c) unsafe $\{T_1{}^+, T_2{}^+\}$
(d) $DL(T_1, T_2)$

Let us define:

DL(T_1,T_2): Given two locked distributed transactions T_1,T_2 construct the digraph DL(T_1,T_2)=(V,A) such that:

(a) V the vertex set, with vertex x iff both T_1 and T_2 contain a Lx-Ux pair.

(b) A the arc set, with arc (xy) iff (Ly $>_{T1}$ Ux and Lx $>_{T2}$ Uy).

An example of DL(T_1,T_2) is presented in Fig. 5.3(d). From the definition of DL(T_1,T_2) we have that (xy)∈ A iff the *upper-left* corner of the x-rectangle is in the *lower-right* corner formed by the y-rectangle on *all* possible (T_1^+,T_2^+)-planes (see Fig. 5.4). This implies that in every such plane no curve corresponding to a history can pass below the y-rectangle and above the x-rectangle.

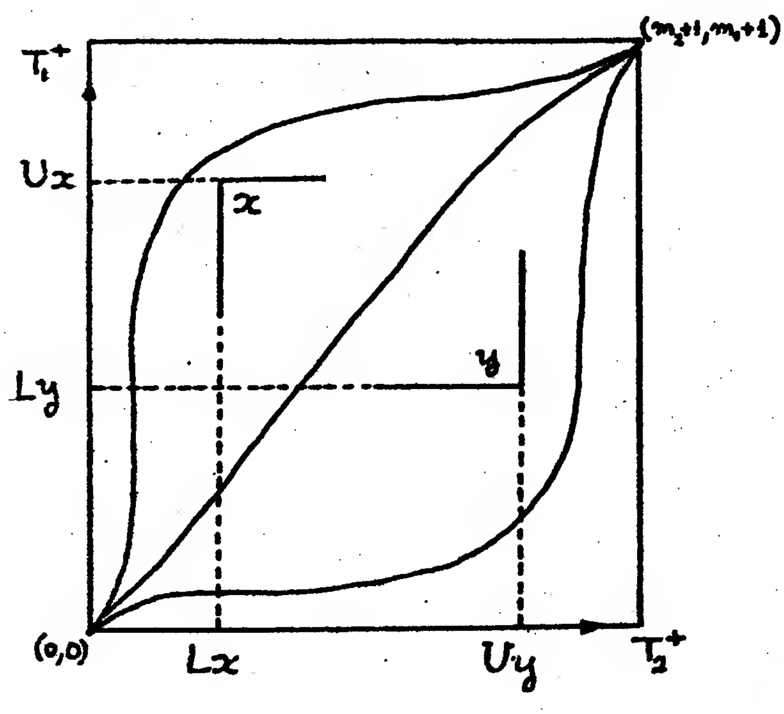


Figure 5.4

(xy)∈ DL(T_1,T_2). Only three types of paths are at most feasible.

Theorem 5: Let $\{T_1,T_2\}$ be a locked transaction system. If $DL(T_1,T_2)$ is strongly connected, then $\{T_1,T_2\}$ is safe.

Proof: Let T_1 and T_2 conflict at variables $x_1,x_2,...,x_k$. Then for $DL(T_1,T_2)=(V,A)$ we have $V=\{x_1,x_2,...,x_k\}$.

In a $(T_1{}^+,T_2{}^+)$-plane we can associate every path s, that corresponds to a possible output history of a lock manager, to a vector of k binary values $\underline{s}=(b_1,b_2,...,b_k)$. These values are:

$b_i=1$ if s passes above the x_i-rectangle

$b_i=0$ if s passes below the x_i-rectangle

Therefore if $(x_i x_j) \in A$ we can say that for all $(T_1{}^+,T_2{}^+)$-planes and paths $\underline{s}$ $b_i \leq b_j$ (i.e. only $b_i=1$, $b_j=1$ or $b_i=0$, $b_j=0$ or $b_i=0$, $b_j=1$ are allowed).

Since $DL(T_1,T_2)$ is strongly connected there is a directed path $(x_i...x_j)$ and a directed path $(x_j...x_i)$ for $1\leq i,j\leq k$, $i\neq j$. Thus always $b_i\leq ...\leq b_j$ and $b_j\leq ...\leq b_i$ for all i,j. This implies that the only allowable values for the vectors $\underline{s}$ are (0,0,...,0) and (1,1,...,1). Thus for all $(T_1{}^+,T_2{}^+)$-planes there is no path corresponding to a history separating two rectangles. Therefore $\{T_1,T_2\}$ is safe.□

In order to characterize safety of a distributed system we need a succinct way of describing the forbidden regions in all $(T_1{}^+,T_2{}^+)$-planes. We use this characterization (as in the proof of Theorem 5) to produce a short proof, that all paths, which correspond to output histories of a lock manager, must either pass below or above all forbidden regions.

The simple condition of safety provided by Theorem 5 is a sufficient one. It is necessary for centralized transactions (Lemma 2), where another obvious complete characterization is the geometric pattern on the unique $(T_1{}^+,T_2{}^+)$-plane. It is also a necessary characterization for transactions distributed between two sites (Theorem 6). Recall that the safety question is in *co-NP*, whereas its negation is in *NP*, that is to prove a system unsafe all we have to do is guess a nonserializable history in O(L) and verify that fact in polynomial time.

We should point out that $DL(T_1,T_2)$ ignores some of the precedences of T_1 and T_2. This restricts the proof of necessity to two sites and indicates that a complete

characterization of forbidden regions for an arbitrary number of sites could be a hard problem.

If $DL(T_1,T_2)=(V,A)$ is not strongly connected then it has more than one strongly connected components. Among these there is a strongly connected component with no incoming edges from other strongly connected components. We call such a component a *dominator* X, where $X \subseteq V$ denotes its set of nodes. In fact the only property of the dominator we will use is that there are no incoming edges in X from nodes in $V \setminus X$ (and not its strong connectivity).

We will prove necessity of the condition in Theorem 5 using the following intuitive construction. Given T_1,T_2, $DL(T_1,T_2)=(V,A)$ not strongly connected and a dominator X, we will construct two special total orders T_1^+,T_2^+. In T_1^+ the actions (Lx-Ux, $x \in X$) will be executed as late as possible after the actions (Lz-Uz, $z \notin X$). In T_2^+ we do the opposite. This tends to isolate the forbidden region corresponding to X in the upper left corner. Each time we will argue that this region and all other rectangles can be separated as in Fig. 5.5, by a curve which will obviously correspond to a possible output history. Therefore we will prove something stronger than lack of safety namely: "If X is such that there are no incoming edges in X, then we can separate all x-rectangles from all z-rectangles, $x \in X$, $z \in V \setminus X$".

<u>Lemma 2</u>: Given a locked transaction system $\{T_1,T_2\}$, where T_1,T_2 are totally ordered, if $DL(T_1,T_2)$ is not strongly connected then $\{T_1,T_2\}$ is unsafe.

<u>Proof</u>: Obviously there is only one (T_1^+,T_2^+)-plane. Pick a dominator X in $DL(T_1,T_2)$. By Theorem 5 all its rectangles form a region that is above an increasing curve, whose corners correspond to lower right corners of x_i-rectangles, $x_i \in X$ (see Fig. 5.6). Let $z \notin X$, then the z-rectangle must be below that curve. If it is not there is an $x_i \in X$ such that $Lz >_{T2} Ux_i$ and $Lx_i >_{T1} Uz$ (since T_1, T_2 are totally ordered) implying that $(zx_i) \in DL(T_1,T_2)$ a contradiction.□

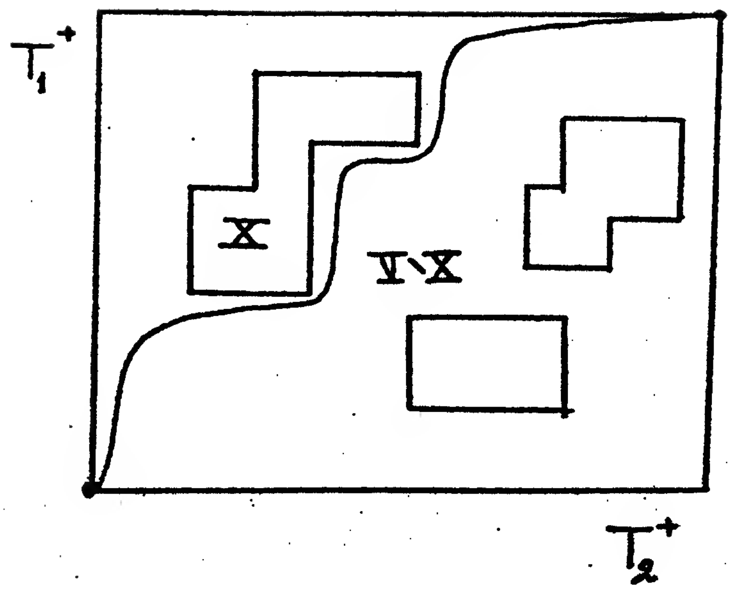


Figure 5.5

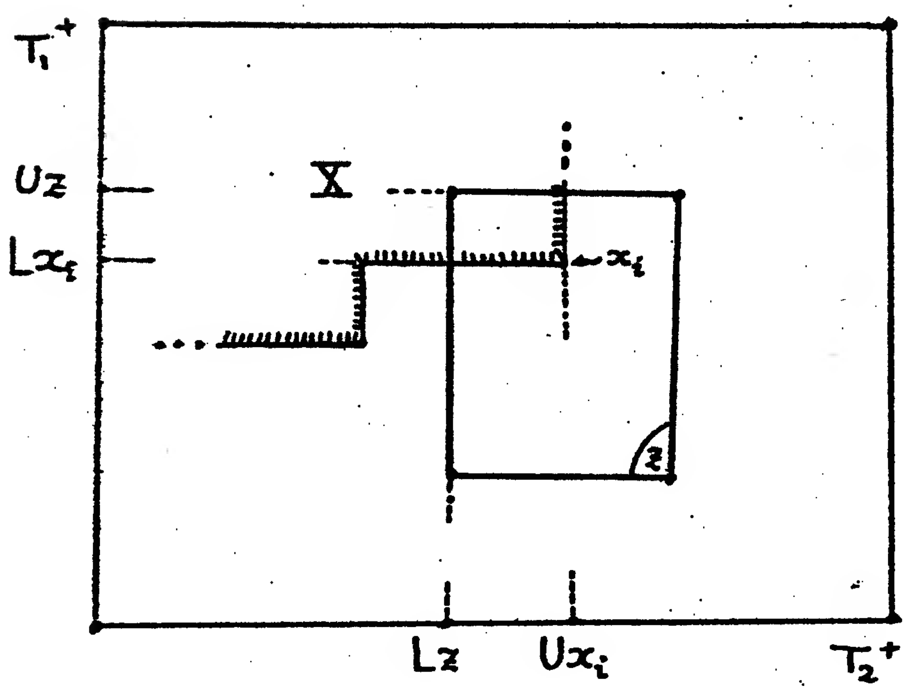


Figure 5.6

Theorem 6: Let $\mathcal{T}=\{T_1,T_2\}$ be a locked transaction system, where T_1,T_2 are distributed at two sites. If $DL(T_1,T_2)$ is not strongly connected then $\mathcal{T}$ is unsafe.

Proof: For this type of distributed transactions there could be an exponential number of possible (T_1^+,T_2^+)-planes. Let X be a dominator of $DL(T_1,T_2)$. We use X to construct two special total orders T_1^+,T_2^+ that will help us separate all x-rectangles ($x\in X$), from all z-rectangles ($z\notin X$) and, since X and $V\setminus X$ are nonempty, this will provide us with a certificate of unsafeness. We will use the shorter notation $>_i$ instead of $>_{Ti}$ and $\gtrdot_i$ for "precedes or can be concurrent to in transaction T_i".

Let z, x, y be such (if they exist) that:

(1) $z\notin X$ and $x,y\in X$

(2) $Lz >_2 Ux$ and $Ly >_1 Uz$

Then we can infer:

(3) $x\neq y$ and $Uy \gtrdot_2 Ux$ and $Uy \gtrdot_1 Ux$.

Since X is a dominator of $DL(T_1,T_2)$ it cannot contain either of the directed edges (zx) or (zy). We can infer (3) because, if $x=y$ $(zx)\in DL(T_1,T_2)$, or if $(Ux >_2 Uy)$ then $(Lz >_2 Uy)$ and $(zy)\in DL(T_1,T_2)$, or finally if $(Lx >_1 Ly)$ then $(Lx >_1 Uz)$ and $(zx)\in DL(T_1,T_2)$.

For any z, x, y satisfying (1),(2) and (3) we can construct the following partial orders:

T_1' is T_1 with the added precedence $Ly >_{1'} Lx$

T_2' is T_2 with the added precedence $Uy >_{2'} Ux$

Obviously T_i' (i=1,2) are partial orders. Also T_i' is T_i (i=1,2) with at most one precedence added (i.e., if the additional precedence were already in T_i then $T_i'=T_i$). Therefore if $\{T_1',T_2'\}$ is unsafe so is $\{T_1,T_2\}$.

Based on the existence of only two sites we will prove the following important fact about the new system $\mathcal{T}'=\{T_1',T_2'\}$:

(I) X is a dominator of $DL(T_1',T_2')$

Since x, y, z are distinct variables we have three cases; case (a) x,y stored at the same site, case (b) x,y stored at different sites and z stored at the same site as x, case (c) x,y stored at different sites and z stored at the same site as y.

Case (a): If x,y are stored at the same site we must have ($Ly >_1 Lx$) and ($Uy >_2 Ux$) (these actions cannot be concurrent in T_1 or T_2). Therefore $T_i' = T_i$ ($i=1,2$) and (I) follows trivially.

Case (b): We have that x and z are stored at the same site and ($Lz >_2 Ux$) (the possible positions of Lz are illustrated in Fig. 5.7). Since $(zx) \notin DL(T_1,T_2)$ we must have ($Uz >_1 Lx$) (i.e. these actions cannot be concurrent in T_1, because x and z are at the same site). Since ($Ly >_1 Uz >_1 Lx$), we have that already ($Ly >_1 Lx$) and therefore $T_1' = T_1$. We only add precedence ($Uy >_{2'} Ux$) to T_2 to obtain T_2'.

The only way for new edges to be generated in $DL(T_1',T_2')$ from a $z' \notin X$ into a $x' \in X$, is for ($Lz' >_2 Uy$) and ($Ux \geq_2 Ux'$) (x' could be x). Moreover z' and x' should be stored at different sites (otherwise Lz',Ux' would have been ordered already in T_2) and in $T_1 = T_1'$ we must have ($Lx' >_1 Uz'$).

If z' and x were stored at the same site, x' must be stored at the site of y. Thus in T_2 we must have had ($Lz' >_2 Uy$ and $Uy >_2 Ux'$) (otherwise the new edge would have introduced a cycle in T_2'). Therefore Lz' and Ux' were already ordered in T_2, a contradiction.

If z' and y were stored at the same site, x' must be stored at the other site and Fig. 5.7 illustrates the possible positions of Lz' and Ux' in T_2. From these ranges of Lz' and Ux' in T_2, we can derive the possible positions of Uz' and Lx' in T_1. Since $DL(T_1,T_2)$ cannot contain either (z'y) or (zx') and since ($Lz' >_2 Uy$ and $Lz >_2 Ux'$), we must have ($Uz' >_1 Ly$ and $Uz >_1 Lx'$). It easily follows from the established ranges that T_1 contains a cycle (UzLx'Uz'LyUz) a contradiction.

This proves (I) for this case.

Case (c): This case is symmetric with case (b). The argument that proves (I) is similar to the one above. The ranges of Lz', Uy' in T_2 and Uz', Ly' in T_1 are illustrated in Fig. 5.8. This time the additional precedence is ($Ly >_{1'} Lx$), and $z' \notin X$, $y' \in X$, and z' must be stored at the site of x, and y' at the site of y.

This completes the proof of (I).

Starting from $\mathcal{T}$ we can construct a sequence of transaction systems $\mathcal{T}', \mathcal{T}'', \ldots, \mathcal{T}^*$ (of length polynomial in $|\mathcal{T}|$) such that in $\mathcal{T}^*$:

(i) X is a dominator of $DL(T_1^*,T_2^*)$

(ii) If ($z \notin X$), ($x,y \in X$), ($Lz >_{2*} Ux$), ($Ly >_{1*} Uz$) then ($Uy >_{2*} Ux$), ($Ly >_{1*} Ux$).

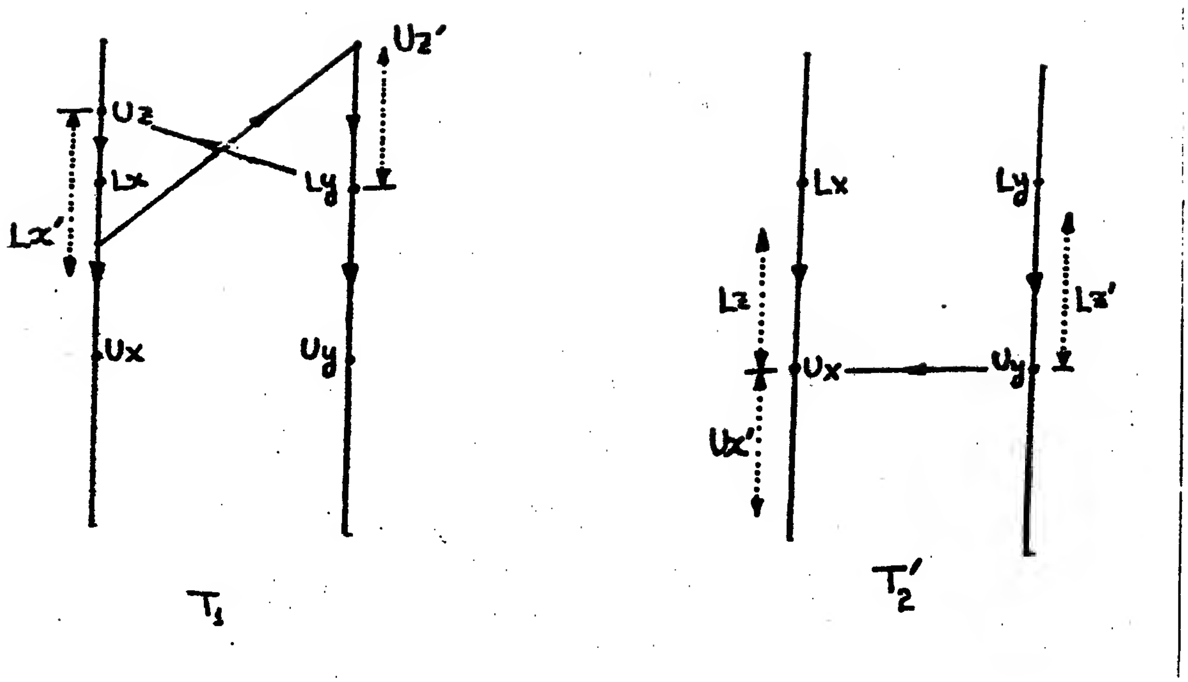


Figure 5.7
Case (b)

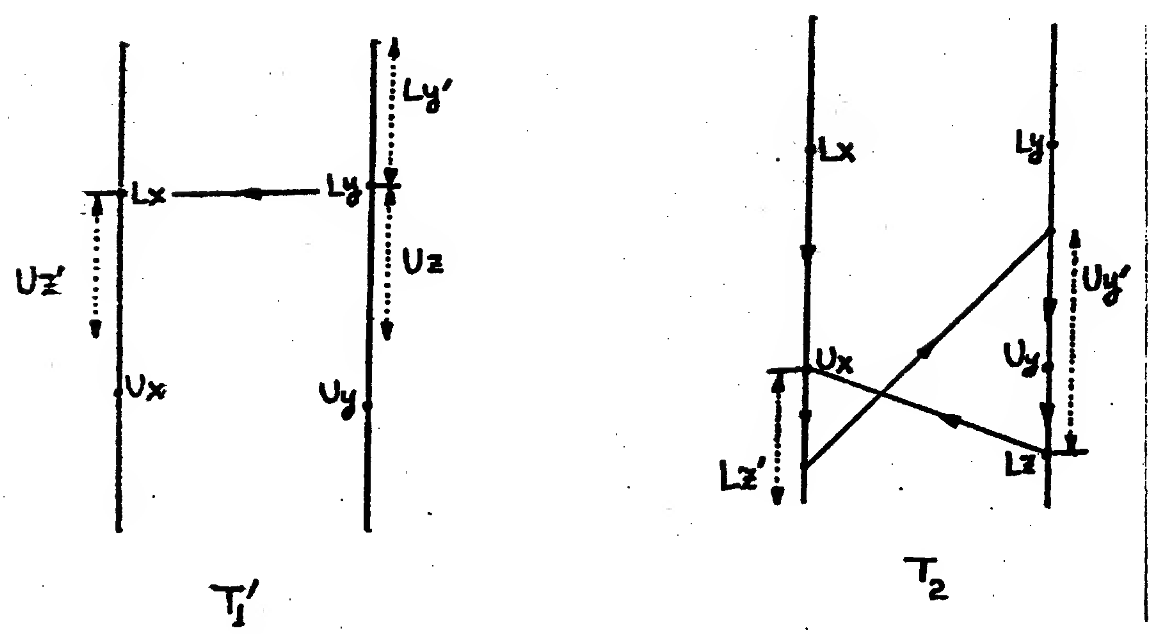


Figure 5.8
Case (c)

Now all we have to do is produce the total orders $T_1{}^+$, $T_2{}^+$ from topologically sorting $T_1{}^*$, $T_2{}^*$. We use two tricks First, we place the Ux (i.e. x in X) steps as *early* as possible in $T_2{}^+$. Second, we place the Lx (i.e. x in X) steps as *late* as possible in $T_1{}^+$, moreover if Ux is before Ux' in $T_2{}^+$ we put Lx before Lx' in $T_1{}^+$ (if possible).

It is easy to see that a nondecreasing curve lower-bounding the area of the rectangles in X is created. Also if (Ly $>_{1+}$ Uz) for some $z \notin X$, and Ly forms part of this curve and is closest to Uz (see Fig. 5.9) then we can easily prove that (Ly $>_{1*}$ Uz). (From the way $T_1{}^+$ was constructed, if there is a closer ($Ly_c >_{1*}$ Uz) we must have (Ly $>_{1*}$ Ly_c) else Ly_c would have been scheduled before Ly in $T_1{}^+$). From the properties of T^* we know that for all $x \in X$ such that (Lz $>_{2*}$ Ux) we have (Uy $>_{2*}$ Ux). By the way $T_2{}^+$ was constructed (Uy as early as possible) we can infer (Uy $>_{2+}$Lz).

Therefore z-rectangles are below or to the left of all x-rectangles in the ($T_1{}^+$,$T_2{}^+$)-plane. This completes the proof of Theorem 6.□

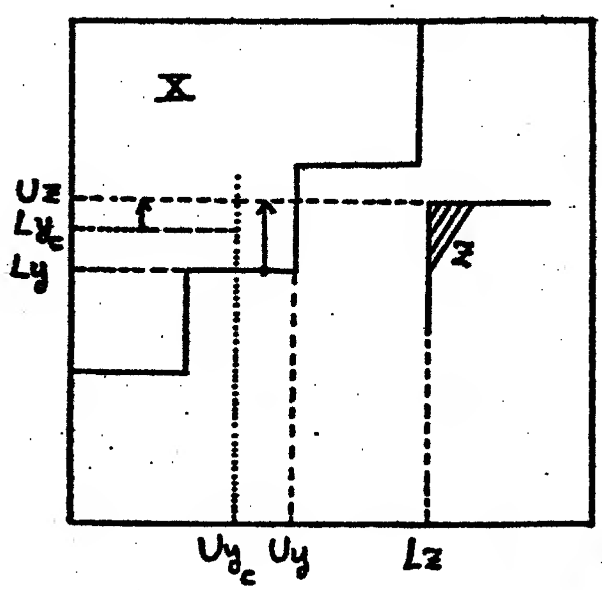


Figure 5.9

The condition of Theorem 6 cannot be applied to systems $\{T_1,T_2\}$ distributed at more than two sites. An example demonstrating that fact is illustrated in Fig. 5.10, where although we have a dominator $X=\{x_1,x_2\}$ (Fig. 5.10(a)) we cannot separate it from the other rectangles (Fig. 5.10(b) and (c)).

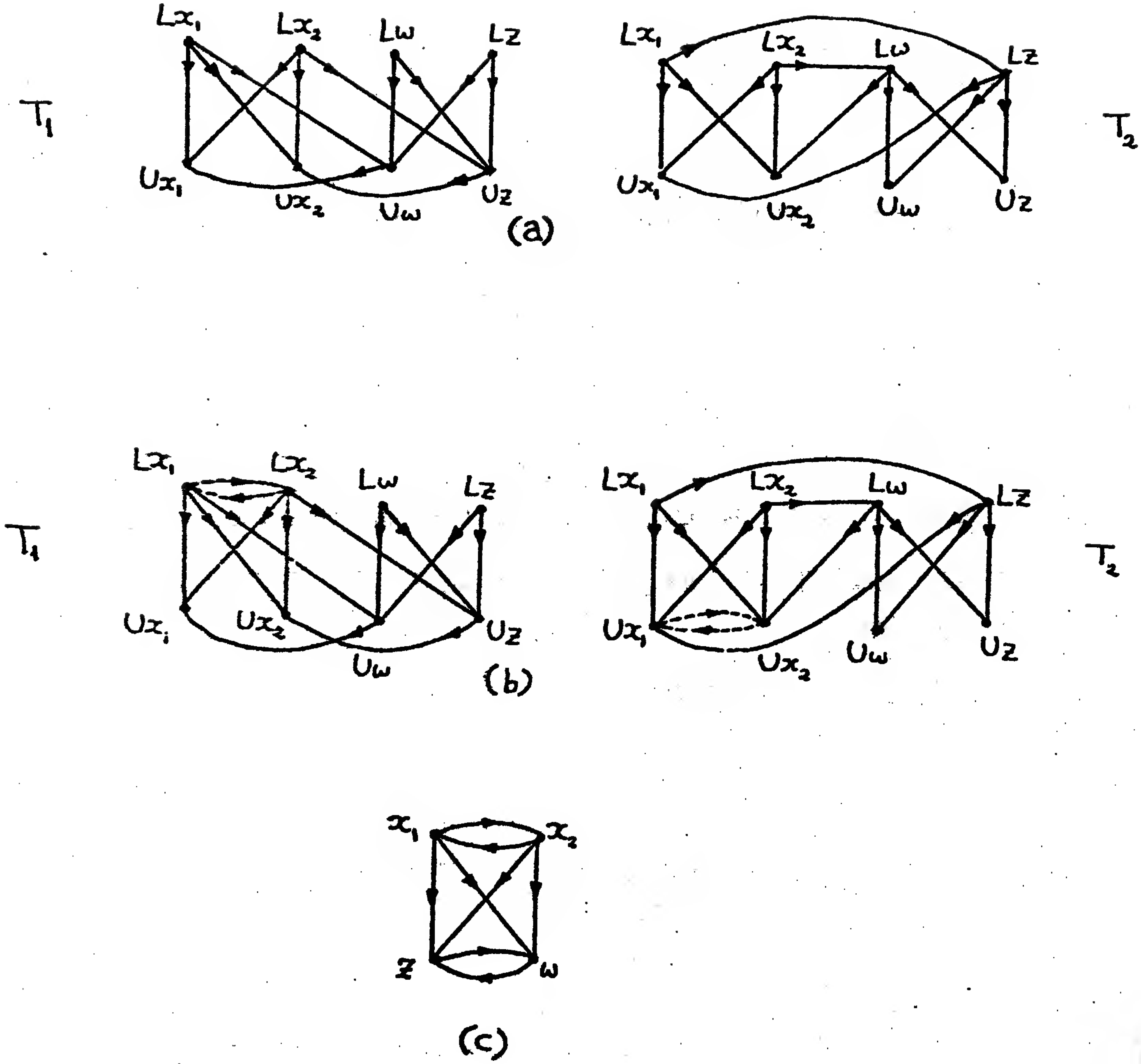


Figure 5.10

(a) T

(b) T^* is not a transaction system

(c) DL(T) has dominator $\{x_1, x_2\}$

Thus we can test safety of distributed transaction systems $T=\{T_1,T_2\}$, on two sites in $O(n^2)$ time [1]. In fact the proof of Theorem 6 gives us the following nondeterministic polynomial time algorithm to decide if an arbitrary system T is unsafe.

Algorithm UNSAFE: Given $T=\{T_1,T_2\}$ a locked transaction system.

(1) Guess a (nonempty) set of rectangles, X that are above a curve, which corresponds to a nonserializable history. Let Z be the (nonempty) set of the rest of the rectangles.

(2) Start with $T_1^*=T_1$, $T_2^*=T_2$ and keep augmenting them by the following rule:
If $z\in Z$, $x,y\in X$, $(Lz >_{2*} Ux)$, $(Ly >_{1*} Uz)$ then add $(Uy >_{2*} Ux)$, $(Ly >_{1*} Lx)$.

(3) Check if T_1^*, T_2^* are partial orders and if $DL(T_1^*, T_2^*)$ has no edges (zx) for $z\in Z$, $x\in X$.

(4) If (3) is true say yes.

The nondeterministic choice at step (1) indicates that the decision problem "Given $T=\{T_1,T_2\}$ is it safe?" may be ***co-NP-Complete***. Such a result would be interesting since it would illustrate the effect of multiple sites on the complexity of the problem.

Until now we have discussed transaction systems T with two transactions. The question of safety of a system with an arbitrary number of centralized transactions is ***co-NP-Complete*** [39], because of a combinatorial condition introduced by the conflict graph $G(T)$. Since the question of safety of a system of an arbitrary number of distributed transactions is in ***co-NP***, we cannot hope to indicate a difference between centralized and distributed by further pursuing this problem.

Another interesting issue is that of deadlock freedom. For the centralized case the geometric approach used for safety [39] gives us a test of deadlock freedom at no extra cost. The approach using $DL(T_1,T_2)$ does not have this nice property.

Therefore we have determined three interesting open problems:

(a) Given a system $\{T_1,T_2\}$ of arbitrary locked distributed transactions, is it safe?

(b) Can the polynomial time bounds implied by Theorems 5 and 6 be improved using the special structure of $DL(T_1,T_2)$?

(c) Given a system $\{T_1,T_2\}$ of locked distributed transactions, is it deadlock-free (even if two sites are used and the system is safe)?

6. Conclusions and Open Problems

We have examined the complexity of distributed database concurrency control. We have provided a rigorous mathematical framework for the study of on-line distributed problems (Chapter 2), established a connection between distributed computation and combinatorial games (Chapter 3) and finally derived both negative (Chapter 4) and positive (Chapter 5) complexity results.

Our main result (Theorem 4) shows that concurrency control, an on-lne problem clearly in ***NP*** in the centralized case, is ***PSPACE-Complete*** in the distributed case. This result is quite strong, in that it holds for transaction systems of rather ordinary appearance (e.g., transactions consisting of sequences of six updates each). Also, the negative implications of our result (Corollary 4.4) are quite robust. For example, even if the scheduler is equipped with a powerful oracle belonging anywhere in the polynomial hierarchy, it still cannot minimize communication efficiently, unless the polynomial hierarchy collapses.

In the process of proving this negative result, we have related distributed concurrency control to certain combinatorial games played on graphs. It could be that this connection is of some practical value, since the length of these games corresponds to counting messages. There is a more-or-less immediate heuristic for approximating an optimal strategy in the game CONFLICT. This heuristic is based on the following purely combinatorial problem, which is still open:

> (I) "Given an undirected graph with its edges colored *red* and *green*, find the smallest set of edges that have to be deleted in order for the resultig graph to have no *two-color* cycle."

Other open problems from Chapter 4 are related to technical issues (II)&(III) or to the messages v.s. computation steps argument of Corollary 4.4 (IV)&(V). This last argument seems quite general in the context of distributed computation.

> (II) Given T without cross-edges and $b \geq 0$ is the minimax length of PREFIX($\langle T, \emptyset \rangle$) greater than b? (conjectured to be ***PSPACE-Complete***)

> (III) Given T is the minimax length of PREFIX($\langle T, \emptyset \rangle$) greater than 0? (conjectured to be in P)

(IV) What are the consequences of $NP = PSPACE$ on the existence of efficient schedulers?

(V) Can a contradiction similar to Corollary 4.4 be derived if $[<T,\emptyset> \notin M_c(b)?]$ is *NP-Complete.*

In Chapter 5 a new $O(n^2)$ safety test was derived for two-transaction locked systems $\{T_1,T_2\}$. This is a necessary and sufficient condition, if transactions are distributed at two sites, and sufficient otherwise. There are a number of interesting open problems.

(VI) Given $\{T_1,T_2\}$ distributed at an arbitrary number of sites are they safe? (conjectured to be ***co-NP-Complete***)

This would demonstrate the complexity introduced by the number of sites.

(VII) Given $\{T_1,T_2\}$ distributed at two sites and safe, are they dead-lock free?

Issues of local and global deadlocks and message-efficient deadlock managers recall the analysis of Chapters 3 and 4.

(VIII) Can the polynomial bounds of $O(n^2)$ (n is number of nodes of the digraph DL) implied by Theorems 5 and 6 be improved using the special structure of DL?

This is possible in the O(nlognloglogn) centralized case.

Finally our analysis of distributed locking can serve as the basis for the development of novel distributed locking strategies, which are not simply generalizations of centralized rules.

References

[1] Aho, A.V., Hopcroft, E., Ullman, J.D. "The Design and Analysis of Computer Algorithms" Addison-Wesley, (1975)

[2] Bernstein, P.A., Rothnie, J.B., Goodman, N. and Papadimitriou, C.H. "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Trans. on Software Eng., vol. SE-4, no. 3 (1978)

[3] Bernstein, P.A., Shipman, D.W., Rothnie, J.B. "Concurrency Control in a System of Distributed Databases (SDD-1)" ACM-TODS, vol. 5, no. 1, (1980)

[4] Bernstein, P.A., Goodman N. "Fundamental Algorithms for Concurrency Control in Distributed Database Systems" Tech. Report, Computer Corporation of America, (Feb. 1980)

[5] Chandra, A.K., Stockmeyer, L.J. "Alternation" Proc. 17th FOCS Conference, pp.98-108, (1976)

[6] Coffman Jr., E.G., Denning P.J. "Operating Systems Theory" Prentice-Hall, (1973)

[7] Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", CACM, vol. 19, no. 11, (Nov. 1976)

[8] Even, S., Tarjan, R.E. "A Combinatorial Problem which is Complete in Polynomial Space" JACM, vol. 23, pp.710-719, (1976)

[9] Feldman, J. "A Programming Methodology for Distributed Computing (among other things)" Tech. Report TR9, Dept. of Computer Science, Univ. of Rochester, (1977)

[10] Garcia-Molina, H. "Performance of Update Algorithms for Replicated Data in a Distributed Database", Ph.D. Dissertation, Computer Science Department, Stanford Univ., (June 1979)

[11] Garey, M.R., Johnson, D.S. "Computers and Intractability: A Guide to the Theory of NP-Completeness" Freeman, (1978)

[12] Gouda, M.G., Dayal U. "Optimal Semijoin Schedules for Query Processing in Local Distributed Database Systems" Proc. ACM-SIGMOD, pp.164-175, (1981)

[13] Gray, J.N., Lorie, R.A., Putzulo, G.R. and Traiger, I.L. "Granularity of Locks and Degrees of Consistency in a Shared Database" IBM Research Report RJ1654, (Sept. 1975)

[14] Hammer, M.M., Shipman, D.W. "Reliability Mechanisms for SDD-1: A System for Distributed Databases" Tech. Report CCA-79-05, Computer Corporation of America (1979)

[15] Hoare, C.A.R. "Communicating Sequential Processes" CACM, vol. 21, no. 8, pp.666-677, (1978)

[16] Kanellakis, P.C., Papadimitriou, C.H. "The Complexity of Distributed Concurrency Control" Proc. 22nd FOCS Conference, (1981)

[17] Kung, H.T., Papadimitriou, C.H. "An Optimality Theory of Database Concurrency Control" Proc. ACM-SIGMOD, pp.116-126, (1979)

[18] Ladner, R.E. "The Complexity of Problems in Systems of Communicating Sequential Processes" Proc. 11th ACM-STOC, pp.214-223, (1979)

[19] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System", CACM, vol. 21, no. 7, pp.558-565, (July 1978)

[20] Lin, W.K. "Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed Database system" Proc. ACM-SIGMOD, pp.84-92 (1981)

[21] Lipski Jr., W., Papadimitriou C.H. "A Fast Algorithm for Testing for Safety and Deadlocks in Locked Transaction Systems", Proc. CISS Conference, Princeton (1980)

[22] Menasce, D.A., Muntz, R.R. "Locking and Deadlock Detection in Distributed Databases", IEEE Trans. on Software Eng., vol. SE-5, no. 3, pp.195-202, (May 1979)

[23] Menasce, D.A., Popek, G.J., Muntz, R.R. "A Locking Protocol for Resource Coordination in Distributed Databases" Proc. ACM-SIGMOD, (1978).

[24] Milne, G., Milner R. "Concurrent Processes and their Syntax" Tech. Report, Univ. of Edinburgh, (1977)

[25] Papadimitriou, C.H. "Serializability of Concurrent Updates" JACM, vol. 26, no. 4, pp. 631-653, (Oct. 1979)

[26] Papadimitriou, C.H. "On the Power of Locking" Proc. ACM-SIGMOD, pp.148-154, (1981)

[27] Reed, D.P. "Naming and Synchronization in a Decentralized Computer System" Ph.D. thesis, M.I.T. Department of EECS, (Sept. 1978)

[28] Rosenkrantz, D.J., Stearns, R.E., Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems" ACM-TODS, vol. 3, no. 2, pp.178-198, (1978)

[29] Schaefer, T.G. "Complexity of Some Perfect Information Games" JCSS, vol. 16, pp.185-225, (1978)

[30] Silberschatz, A. Kedem, Z. "Consistency in Hierarchical Database Systems" JACM, vol. 27, no. 1, pp.72-80 (Jan. 1980)

[31] Stearns R.S., Lewis, P.M., Rosencrantz, D.J. "Concurrency Control for Database Systems" Proc. 16th FOCS Conference, pp.19-32, (1976)

[32] Stearns R.S., Rosencrantz, D.J. "Distributed Database Concurrency Control Using Before-Values" Proc. ACM-SIGMOD, pp.74-83, (1981)

[33] Stockmeyer, L.J., "The Polynomial-time Hierarchy" Theor. Computer Sci., 3, pp.1-22, (1976)

[34] Stockmeyer, L.J., Meyer, A.R. "Word Problems Requiring Exponential Time" Proc. 5th ACM-STOC, pp.1-9, (1973)

[35] Stonebraker, M. "Concurrency Control and Consistency of multiple Copies of Data in Distributed INGRES" IEEE Trans. on Software Eng., vol. SE-5, no. 3, pp.188-194, (May 1979)

[36] Thomas, R.H. "A Majority Consensous Approach to Concurrency Control for multiple Copy Databases" ACM-TODS, vol. 4, no. 2, pp.180-209, (1979)

[37] Ullman, J.D. "Principles of Database Systems" Computer Science Press, (1980)

[38] Yao, A.C. "Some Complexity Questions Related to Distributive Computing" Proc. 11th ACM-STOC, pp. 209-213, (1979)

[39] Yannakakis, M., Papadimitriou, C.H., Kung, H.T. "Locking Policies: Safety and Freedom from Deadlock" Proc. 20th FOCS Conference, pp.283-287, (1979)

[40] Yannakakis, M. "Issues of Correctness in Database Concurrency Control by Locking" Proc. 13th ACM-STOC, pp.363-367, (1981)

Index of Terms

Index of Figures and Tables

Biographical Note

The author was born on December 3, 1953, in Athens, Greece, where he lived until 1976. He attended the National Technical University of Athens and received a Diploma in Electrical Engineering with honors in June 1976. The following September he started his graduate work in the Department of Electrical Engineering and Computer Science at M.I.T.. He completed his M.S. degree in Electrical Engineering and Computer Science in June 1978 and his Ph.D. degree in Computer Science in September 1981.

The author will join the Computer Science faculty of Brown University as an assistant professor.

CS-TR Scanning Project
Document Control Form

Date : 8/18/95

Report # LCS-TR-269

Each of the following should be identified by a checkmark:
Originating Department:

- [ ] Artificial Intellegence Laboratory (AI)
- [x] Laboratory for Computer Science (LCS)

Document Type:

- [x] Technical Report (TR)
- [ ] Technical Memo (TM)
- [ ] Other: ____________

Document Information

Number of pages: 122 (129-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
- [x] Single-sided or
- [ ] Double-sided

Intended to be printed as :
- [ ] Single-sided or
- [x] Double-sided

Print type:
- [ ] Typewriter
- [ ] Offset Press
- [ ] Laser Print
- [ ] InkJet Printer
- [x] Unknown
- [ ] Other: ____________

Check each if included with document:
- [ ] DOD Form
- [ ] Funding Agent Form
- [x] Cover Page
- [x] Spine
- [x] Printers Notes
- [ ] Photo negatives
- [ ] Other: ____________

Page Data:

Blank Pages(by page number): FOLLOW TITLE PAGE, i, ii, iii, 103

Photographs/Tonal Material (by page number): ____________

Other (note description/page number):

Description : Page Number:

IMAGE MAP (1-8) UN#'ED TITLE PAGE, UN#'ED BLANK, i, UN#'ED BLANK, ii, UN#'ED BLANK, iii, UN#'ED BLANK
(9-122) PAGES #'ED 1-103, UN#'ED BLANK, 104-113
(123-129) SCANCONTROL, COVER, SPINE, PRINTER'S NOTES, ~~[illegible]~~ TRGT'S (3)

Scanning Agent Signoff:

Date Received: 8/18/95 Date Scanned: 9/19/95 Date Returned: 9/21/95

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd